MASTER EN SCIENCES INFORMATIQUES

État de l'art en matière de contrôle de concurrence pour bases de données

Lampe, Dominique

Award date:
1988

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
NAMUR
Institut d'Informatique

Etat de l'art en matière de contrôle de concurrence pour bases de données

Dominique Lampe

Mémoire présenté en vue de l'obtention du titre de
Licencié et Maître en Informatique

1987 - 1988

FACULTÉS
UNIVERSITAIRES
N. D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Etat de l'art en matière de contrôle de concurrence pour bases de données

Promoteur

Monsieur **Jean RAMAEKERS**

Mémoire présenté en vue de l'obtention du titre

de Licencié et Maître en Informatique

par

Dominique LAMPE

Année académique 1987 - 1988

Résumé

Dans ce mémoire, nous présentons un état de l'art en matière de contrôle de concurrence et des algorithmes nécessaires pour synchroniser les transactions de multiples utilisateurs dans des systèmes centralisés et distribués. Dans la première partie de notre travail, nous analysons des protocoles centralisés, en particulier ceux basés sur le verrouillage, mais nous étudions aussi l'estampillage, le test du graphe de sérialisation, les méthodes optimistes et les schedulers intégrés. Nous comparons les différents protocoles utilisés par les schedulers et leur impact sur les transactions dont les opérations sont en conflit. La redondance des données est également envisagée comme une manière d'améliorer l'efficience du système lorsqu'elle est transparente aux utilisateurs. Notre seconde partie traite des problèmes de contrôle de concurrence dans un environnement distribué et de leurs solutions. La redondance des données et l'indépendance des sites de travail sont les principales causes de ces problèmes et nous discutons des possibilités d'adapter les protocoles centralisés à ce type d'environnement. Il n'est pas dans l'optique de notre travail d'en développer les aspects formels ni d'en analyser les impacts au niveau des performances, même si nous tenons parfois compte de ces considérations afin d'améliorer des méthodes de synchronisation existantes.

Abstract

In this dissertation, we present the state of the art of concurrency control techniques imposed to synchronize multiple user transactions in centralized and distributed database systems. In the first part of our work, we analyze centralized protocols, especially those based on locking, but we also study timestamping, serialization graph testing, optimistic methods and integrated schedulers. We compare the different protocols used by the schedulers and their impact on transactions with conflicting operations. User transparent data redundancy is examined as a means to improve efficiency, too. Our second part is dedicated to distributed concurrency control problems and their answers. Data redundancy and independent working sites are the main sources of those problems and we discuss the possibility of adapting centralized protocols to cope with this distributed environment. The aim of our work is neither to develop formal aspects of these techniques nor to analyze their performance impacts, even if we sometimes take them into account in order to improve existing synchronization methods.

Que soient ici remerciés tous ceux qui m'ont, d'une manière ou d'une autre, aidé lors de la réalisation de ce travail.

Je pense tout particulièrement à Monsieur Jean Ramaekers, promoteur de mon mémoire, qui m'a guidé et conseillé au cours de cette année académique.

Toute ma gratitude aussi à son assistante, Madame Cécile Stas-Mahiat, pour les documents qu'elle m'a fournis et pour sa disponibilité à mon égard.

Je tiens enfin à exprimer ma profonde reconnaissance à Monsieur Philippe De Rivet ainsi qu'aux membres du groupement technique de Bull Transac, Paris, pour m'avoir permis d'effectuer mon stage de fin d'études dans les meilleures conditions.

Namur, le 31 août 1988,

Dominique Lampe

PARTIE I
TECHNIQUES DE CONTROLE DE CONCURRENCE POUR BASES DE DONNEES CENTRALISEES

Introduction 1

1. Quelques concepts 2

2. Les problèmes 3

2.1. Les mises à jour perdues ("Lost updates") 5

2.2. Les données instables ("Dirty read") 5

2.3. Les incohérences statistiques ("Inconsistent retrieval") 7

3. Acceptabilité des solutions générales pour un O.S. 7

3.1. Sérialisation des processus 8

3.2. Préemption des processus 8

3.3. Ordre préalable sur les ressources 8

3.4. Revendication des ressources nécessaires 8

Chapitre 1 : Le modèle de travail 10

Chapitre 2 : La sérialisabilité 13

Chapitre 3 : Techniques de verrouillage 15

3.1. Principe de base 15

3.2. Le verrouillage à 2 phases : 2PL 16

3.3. Variantes du 2PL 20

3.3.1. Le 2PL conservateur : la prédéclaration 20

3.3.2. Le 2PL strict 22

3.3.3. Variante profitant des "shadow values" 23

3.4. Conversion de verrous 26

3.5. Les problèmes liés au verrouillage 29

3.5.2. Les "deadlocks" 31

3.5.2.1. Le "timeout" 33

3.5.2.2. Le graphe des mises en attente : WFG 34

3.5.2.3. La détection 35

3.5.2.4. La prévention 38

3.5.3. Les livelocks 44

3.6. La granularité 47
3.6.1. Le verrouillage prédicat 47
3.6.2. Le verrouillage physique 48
3.7. La multi-granularité (ou hiérarchie des verrous) 49
3.7.1. Le mode d'intention 51
3.7.2. L'escalade et la conversion de verrou en MGL 55
3.7.3. Graphe acyclique orienté : DAG 56

Chapitre 4 : L'estampillage : TO 59

4.1. La règle de l'estampillage 59
4.2. Protocole de base 60
4.3. L'estampillage strict 62
4.4. L'estampillage conservateur 63

Chapitre 5 : Le test du graphe de sérialisation : SGT 66

5.1. Le graphe de sérialisation 66
5.2. Protocole de base 68
5.3. Le SGT conservateur 69

Chapitre 6 : Les méthodes optimistes 71

6.1. Principe général 71
6.2. La certification à base de 2PL 73
6.3. La certification à base de TO 73
6.4. La certification à base de SGT 74

Chapitre 7 : Les données à versions multiples 75

7.1. Verrouillage à deux phases multiversion 76
7.1.1. 2PL pour deux versions : 2V2PL 76
7.1.2. 2PL pour plus de deux versions : MV2PL 78
7.2. Estampillage multiversion : MVTO 79

Chapitre 8 : Verrouillage vs non-verrouillage 83

8.1. Avantages et inconvénients du verrouillage 83

8.2. Avantages et inconvénients de l'estampillage 84

8.3. Avantages et inconvénients du test de sérialisation 84

8.4. Avantages et inconvénients des méthodes optimistes 85

8.5. Les schedulers intégrés 85

8.5.1. La règle d'écriture de Thomas : TWR 87

8.5.2. Scheduler intégré pur 89

8.5.3. Scheduler intégré mixte 89

Conclusion 91

PARTIE II
TECHNIQUES DE CONTROLE DE CONCURRENCE POUR BASES DE DONNEES DISTRIBUEES

Introduction 93

Chapitre 9 : Le modèle de travail 95

Chapitre 10 : La redondance des données 97

10.1. L'approche Write-All 99

10.2. L'approche Write-All-Available 100

Chapitre 11 : Le verrouillage à deux phases : D2PL 101

11.1. Version D2PL sans redondance 102

11.2. Versions du D2PL avec redondance 103

11.2.1. Version de base du D2PL 103

11.2.2. D2PL Centralisé 103

11.2.3. D2PL avec Copies Principales 104

11.2.4. Algorithme des copies disponibles : ACA 104

11.3. Les interblocages dans les SGBDD 107
11.3.1. Détection des interblocages 108
11.3.1.1. Approche centralisée 108
11.3.1.2. Approche hiérarchique 109
11.3.2. Prévention des interblocages 110
11.3.2.1. Implémentation en D2PL 110
11.3.2.2. Les méthodes prioritaires 111
11.3.2.3. L'estampillage et les problèmes de synchronisation 111
11.3.2.4. Des alternatives 112

Chapitre 12 : L'estampillage : DTO 113

12.1. Protocole DTO de base 113
12.2. Version conservatrice du DTO 114

Chapitre 13 : Test du graphe de sérialisation : DSGT 116

Chapitre 14 : Les méthodes optimistes et le vote 118

Chapitre 15 : Alternatives 119

15.1. Pour les données non redondantes 119
15.2. Pour les données redondantes 120

Conclusion 121

INTRODUCTION

De nombreuses méthodes destinées à contrôler les accès concurrentiels ont déjà été créées, les premières destinées aux systèmes d'exploitation, les plus récentes aux bases de données. La prolifération des systèmes multi-utilisateurs, des bases de données communes ou des réseaux locaux explique l'importance considérable qu'une gestion correcte des accès aux données peut avoir. Tout en cherchant à accroître le parallélisme d'accès aux bases de données et, par conséquent, la satisfaction des clients, les concepteurs des systèmes de gestion de bases de données cherchent à diminuer leurs propres coûts de fonctionnement. Ils chercheront dès lors tous à trouver La ou Les méthodes s'adaptant le mieux à leur configuration. C'est ce qui explique l'apparition en nombre impressionnant de telles méthodes ou de telles variantes.

L'objectif de notre travail est de mettre un peu d'ordre dans cette masse d'algorithmes de contrôle de concurrence pour bases de données. Nous nous sommes basés sur bon nombre d'articles et sur quelques ouvrages de synthèse en matière de contrôle de concurrence et de recouvrement, dont les références bibliographiques sont données en annexe. Nous avons essayé d'en élaguer les aspects formels afin de nous concentrer sur les algorithmes et leur comportement dans diverses situations.

La première partie de notre travail s'intéresse au cas des bases de données centralisées. Nous avons choisi de l'introduire de manière assez complète par un exposé des problèmes justifiant le contrôle de concurrence et des particularités des bases de données par rapport à d'autres ressources.

Dans le chapitre 1, nous proposons le modèle qui servira tout au long de cette partie. Son coeur est un module appelé "scheduler" et qui concentre en lui les protocoles de contrôle de concurrence. Puisqu'il est unique dans un tel environnement, nous avons axé notre travail sur une compréhension approfondie de son fonctionnement. Il est chargé de régler les accès en provenance des transactions des utilisateurs et de les faire exécuter sur la base de données physique.

Dans le chapitre 2, nous survolons une théorie à la base des preuves formelles de validité des schedulers, la théorie de la sérialisabilité.

Le chapitre 3 est le plus important de cette première partie. Il tente d'établir une synthèse des méthodes utilisant le principe du verrouillage. Le verrouillage à deux phases est fondamental dans cette classe d'algorithmes. Nous développons dans ce chapitre des notions importantes comme le conflit, la conversion de verrous, l'interblocages et la famine. Nous nous sommes également intéressés de près au choix des unités de verrouillage, autrement dit à la granularité.

Le chapitre 4 est le premier d'une série de chapitres consacrés aux techniques qui ne sont pas fondées sur le principe du verrouillage. L'estampillage en est l'objet. Cette méthode basée sur une attribution de priorités a des caractéristiques extrêmement intéressantes pour la simplicité de sa mise en oeuvre.

Les chapitres 5 et 6, un peu moins conventionnels, traitent respectivement d'une méthode assez formelle testant un graphe dit "de sérialisation" et d'une méthode qualifiée d'"optimiste" puisque, a priori, elle ne contrôle pas la concurrence.

Le chapitre 7 suggère de multiplier les lieux de stockage d'une même donnée physique afin d'augmenter la concurrence du système.

Le chapitre 8 dresse un tableau comparatif entre les techniques vues aux chapitres 3, 4, 5 et 6 et propose par conséquent une solution mitigée : les schedulers intégrés.

La seconde partie de notre travail est presque parallèle à la première, mais nous mettons plus l'accent sur les difficultés inhérentes à la répartition des sites que sur des algorithmes; elle sera dès lors plus succincte.

Le chapitre 9 propose le modèle d'une telle configuration.

Le chapitre 10 traite de la redondance des données. Celle-ci fait presque partie intégrante des systèmes distribués et conditionne la suite de notre travail.

Les chapitres 11, 12, 13 et 14 sont les équivalents respectifs des chapitres 3, 4, 5 et 6, mais dans un environnement où les sites sont multiples et presque indépendants.

Le chapitre 15 propose quelques alternatives aux méthodes présentées lors des chapitres précédents.

PARTIE I

TECHNIQUES DE CONTROLE DE CONCURRENCE POUR BASES DE DONNEES CENTRALISEES

Introduction

Le problème de l'intégrité des bases de données suscite de l'intérêt chez tous les utilisateurs et souvent de l'inquiétude. Ceux-ci souhaitent à tout moment pouvoir disposer d'une base de données (notée BD) contenant des données exactes et cohérentes, quels qu'aient pu être les problèmes encourus par le système jusqu'à ce moment-là.

Nous supposerons qu'intégrité sémantique et intégrité syntaxique sont déjà assurées par un module du Système de Gestion de Base de Données (noté **SGBD**).

L'hypothèse selon laquelle le SGBD assume l'intégrité syntaxique est assez réaliste. Par contre, pour ce qui est de l'intégrité sémantique, les contraintes peuvent être tellement variées et complexes que le programmeur d'application devra sans doute en gérer une partie lui-même, de façon dynamique. La spécificité et la diversité des domaines d'application d'un SGBD font en effet que des solutions complètes (standardisées et acceptables) n'existent, à notre connaissance, pas encore. ("*La spécification de ces règles d'intégrité est évaluée à quelque 80 % d'une description typique - du schéma conceptuel en fait - d'une base de données.*", [DATE83]).

La littérature parle aussi d'intégrité opérationnelle. Nous signifions par là que, si le module assurant les intégrités syntaxique et sémantique de la Base de Données (notée **BD**) fonctionne correctement, alors il faudra aussi garantir que cette intégrité ne soit pas altérée par une utilisation quelconque du SGBD. Plus explicitement, nous entendons protéger la BD des interférences qu'un travail en environnement multi-utilisateur pourrait éventuellement provoquer. Notre travail présentera les techniques de préservation de ce type d'intégrité et nous verrons qu'en effet, malgré un système de vérification valide, des accès peuvent, a priori, perturber plus ou moins insidieusement le travail du SGBD.

Nous distinguons **deux modes de fonctionnement du SGBD** :

- ◊ le mode normal, pendant lequel les applications s'effectuent proprement, sans faute ni du hardware ni du système,
- ◊ et le mode anormal, qui concerne un SGBD devant traiter de manière appropriée des problèmes tels que la destruction partielle ou totale de la BD physique, l'effondrement complet du système, suite à des problèmes d'alimentation électriques ou encore la terminaison anormale d'une application.

Curtice [CURT77] étudie de plus près les différentes classes d'intégrité liées à chacun de ces deux modes de fonctionnement.

A la classe du mode de fonctionnement normal, il associe les problèmes de contrôle de concurrence (régulation des accès en lecture/écriture) et d'interblocages. Notre travail s'efforcera de présenter les techniques visant à préserver l'intégrité dans ces circonstances. Les problèmes seront généralement introduits à l'aide d'exemples et nous montrerons comment, de raisonnement en raisonnement, on arrive enfin à une solution acceptable (ou non). Nous rappelons que tout cela a lieu en mode normal, c'est-à-dire dans un milieu sain.

A la classe du mode de fonctionnement anormal, Curtice associe les divers problèmes liés aux accidents perturbant le système et les classes d'intégrité correspondantes. Nous n'en parlerons pratiquement pas, à moins que cela ne permette d'éclairer une situation. Les techniques en la matière tiennent de la tolérance aux fautes et ont notamment trouvé leur synthèse dans [CHAN87].

1. Quelques concepts

Nous avons, au moyen des quelques lignes ci-dessus, esquissé le sujet de notre travail. Avant d'en aborder les points essentiels, nous souhaitons présenter les concepts fondamentaux, ceux dont nous nous servirons constamment. Nous cherchons par là à éviter de notre mieux les malentendus qui pourraient découler d'une définition inexistante des notions utilisées, tout en essayant de n'entrer en contradiction avec aucune autre définition existante. Les concepts plus spécifiques à une matière donnée seront introduits lors du développement-même de celle-ci.

Une **base de données** est un ensemble d'objets nommés entre lesquels existent certaines relations qu'on peut présenter comme des assertions sur ces objets.

Des exemples de telles assertions pourraient être : "*Solde_client est la différence entre le montant au débit et celui au crédit d'un client*", "*Num_référence est un index pour les livres de la bibliothèque X*" ou, plus simplement, "$A = B$"

On dit que cette BD est dans un **état cohérent**[1] lorsqu'elle satisfait à toutes ces assertions. Dans certains cas, la BD sera temporairement incohérente afin de permettre une transformation vers un nouvel état cohérent.

[1] Dans la littérature, on distingue parfois la cohérence de l'intégrité. Dans notre travail, nous confondons volontairement les deux notions : pour nous, assurer la cohérence d'une BD signifie que nous garantissons que des erreurs ne seront pas introduites par le fait de l'exécution simultanée des programmes de plusieurs utilisateurs. Les contraintes d'intégrité, quant à elles, sont censées être vérifiées par le SGBD.

Par exemple, lors du transfert d'un certain montant d'un compte en banque vers un autre, il existera un laps de temps pendant lequel le premier compte est déjà débité alors que le second n'est pas encore crédité. A ce moment, l'assertion selon laquelle le volume monétaire lors d'un transfert reste constant est donc violée. Mais cet état n'est que transitoire, car dès que la suite du transfert aura eu lieu, la BD retrouvera un état cohérent.

Afin de faire face à ces incohérences momentanées, des séquences d'actions atomiques[1] sont groupées pour constituer ce qu'on appelle des **transactions**. Ce sont des actions atomiques plus grandes qui font passer le système d'un état cohérent vers un nouvel état cohérent. Si, pour l'une ou l'autre raison, une des actions de la transaction devait échouer, alors tout ce qui a déjà été effectué jusque là dans le cadre de cette transaction doit être "défait" par le SGBD, remettant la BD dans l'état cohérent qui précédait cette tentative de transaction. Les transactions préservent donc la cohérence.

Une transaction peut échouer pour différentes raisons telles que des fautes matérielles ou logicielles, des erreurs système, des interblocages ou des violations de protections, ou simplement sur demande. Dans ce dernier cas, nous dirons que la **transaction** est **avortée** (traduction du verbe anglais "to abort").

Parcontre, si la transaction parvient à s'exécuter jusqu'au bout, alors ses effets seront définitivement répercutés sur la BD. Nous dirons alors que la **transaction** est **confirmée** (traduction de l'anglais "to commit").

2. Les problèmes

Nous nous devons de distinguer deux classes d'opérations d'accès sur la BD : les opérations de simple consultation[2] et les opérations de mise à jour[3].

Il est bien évident que si les données n'étaient accédées qu'en lecture, elles ne seraient soumises à aucun risque de corruption. Un SGBD se chargeant d'une telle configuration n'aurait à effectuer aucun contrôle de concurrence.

De même, si la BD n'était accédée que par une seule personne à la fois, il n'y aurait évidemment aucun problème de gestion d'accès concurrentiels. En effet, si les transactions sont exécutées séquentiellement, alors chaque transaction héritera de son prédecesseur d'une BD cohérente.

---

[1] Des actions atomiques sont des actions qui sont normalement ininterrompues. Ce sont des unités au sens fort du terme.

[2] Dans ce cas, la littérature parle de lecteur.

[3] Dans ce cas, la littérature parle de rédacteur.

Mais la réalité est toute différente : les bases de données sont généralement volumineuses et à la disposition de nombreux utilisateurs travaillant en temps réel. Surgit alors le problème délicat du partage des ressources, malgré le fait que chaque transaction prise isolément conserve la cohérence du système.

Remarque :

Un des critères déterminants lors du choix d'un SGBD sera le nombre de transactions qu'il est capable de traiter en un certain laps de temps. Selon ce critère, les SGBD ne permettant aux transactions que de s'exécuter séquentiellement partent donc avec un sérieux handicap.

On introduit alors la **concurrence** pour améliorer l'utilisation et les temps de réponse d'un tel système. Il s'agit de permettre à plusieurs utilisateurs de travailler en parallèle sur un système commun (BD et/ou autres ressources). Ce parallélisme contrôlé ne devrait donc ni perturber le déroulement des programmes, ni mobiliser plus de ressources (CPU, disques, bus, etc...) qu'il ne permet de "gagner".

Au lieu de techniques de contrôle de concurrence, on parlera aussi parfois de **techniques de synchronisation**. Sauf notification explicite[1], nous utiliserons les deux termes pour désigner les mêmes techniques, celles qui résolvent le problème du contrôle de concurrence dans son ensemble.

Une littérature abondante est consacrée au contrôle de concurrence pour diverses ressources. Notre travail ne s'attachera qu'au cas des bases de données. Nous montrerons cependant au point 3 en quoi celles-ci diffèrent des autres ressources et pourquoi certaines solutions générales pour un système d'exploitation doivent être rejetées.

Si nous envisageons de contrôler la concurrence, c'est qu'il y a des problèmes à résoudre. Et en effet, nous en distinguerons de deux catégories :

- les problèmes de perception incorrecte de la BD;
- les problèmes de destruction de l'intégrité de la BD.

La perception incorrecte de la BD par un lecteur se produit généralement lorsque celui-ci accède à des données qui sont en cours de modification par d'autres transactions. Le problème n'est pas trop grave dans la mesure où il n'est que temporaire, où le lecteur peut éventuellement s'en rendre compte et où ce dernier n'envisage pas de devenir rédacteur sur base des données qu'il vient d'acquérir.

---

[1] Dans la littérature, les méthodes de contrôle de concurrence sont généralement considérées comme celles qui résolvent le problème complet du contrôle de concurrence, alors que les techniques de synchronisation sont plutôt les algorithmes qui résolvent des sous-problèmes du contrôle de concurrence.

Toutefois, le problème de la destruction de l'intégrité de la BD par un rédacteur montre bien que si des transactions modifient des données partagées, alors leurs exécutions concurrentes doivent absolument être régulées.

La source de ces problèmes est localisée dans la découpe, généralement en trois niveaux, de la mémoire :

- le niveau tampon des programmes d'application;
- le niveau tampon du SGBD;
- le niveau physique de la BD.

Cette découpe en niveaux a un but souvent d'amélioration des performances parfois d'interfaçage. Ces objectifs sont à la base des problèmes de contrôle de concurrence car pour les réaliser, on recourt généralement à une désynchronisation entre le niveau de l'application et celui des accès physiques. On parle de désynchronisation parce que les modifications demandées par les transactions ne sont pas répercutées immédiatement sur la BD sur support physique, mais plutôt sur un morceau de celle-ci en mémoire centrale.

Afin d'illustrer les problèmes cités ci-dessus, nous prendrons 3 exemples classiques : celui des mises à jour perdues, celui des données instables et celui des incohérences statistiques. Ces exemples sont inspirés de [BERN81].

2.1. Les mises à jour perdues ("Lost updates")

Le problème de la mise à jour perdue survient lorsque deux transactions lisent une même donnée presque simultanément, la modifient puis l'écrivent l'une après l'autre, la seconde écrasant la donnée à peine écrite par la première. La figure 1 montre comment deux exécutions concurrentes peuvent interférer. L'exécution correcte des deux transactions aurait dû donner, dans ce cas, un compte créditeur de 7000 Francs (F) et non de 9000 ou 3000 F. Il y a bel et bien destruction de l'intégrité de la BD.

2.2. Les données instables ("Dirty read")

Le problème survient lorsqu'une transaction lit une donnée A puis la modifie en A'. A ce moment, une transaction concurrente lit cette donnée A' et continue son exécution. Ensuite, la première transaction doit, pour l'une ou l'autre raison, être défaite (par conséquent, la BD doit contenir la donnée A et non A'). La donnée A' lue par la seconde transaction n'a donc jamais existé. C'est ce que représente la figure 2. La BD restera cohérente sauf si la seconde transaction se sert de la donnée A' (parfois appelée "donnée fantôme") lors d'une mise à jour de la BD.

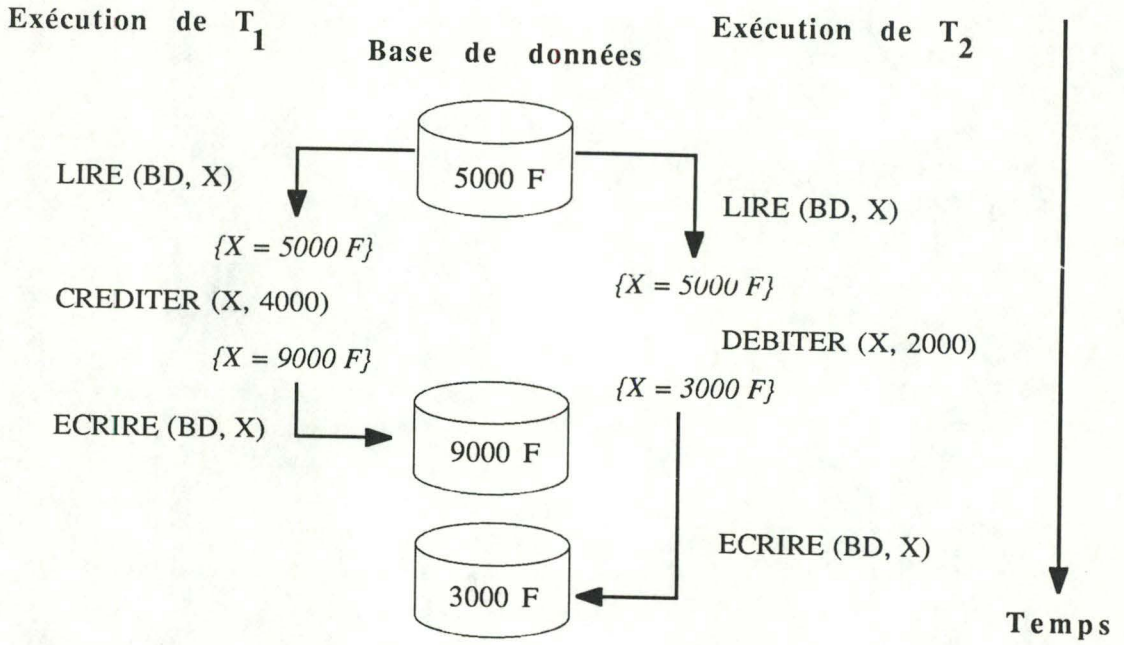


Figure 1 : Mise à jour perdue

Transaction$_1$

début de la transaction

LIRE (BD, X)

{X = A}

CALCULER (X = A')

MODIFIER (BD, X)

{X = A'}

-
-
-

la transaction doit être défaite

{X = A}

Transaction$_2$

début de la transaction

LIRE (BD, X)

{X = A'}

fin de la transaction

Figure 2 : Donnée instable

2.3. Les incohérences statistiques ("Inconsistent retrieval")

Ce troisième exemple est un peu plus particulier, car il montre comment un problème peut surgir entre deux transactions ne travaillant pas nécessairement en même temps sur les mêmes données et qui est quand-même un problème de contrôle de concurrence. La figure 3 est l'illustration d'un tel problème. La première transaction calcule la somme de tous les dépôts; la seconde effectue un transfert de 500 F du compte 2 vers le compte 4. Si la seconde transaction est exécutée après que la première ait lu le montant du compte 2 mais avant qu'elle n'ait lu le montant du compte 4, le total obtenu par la première transaction sera excédentaire de 500 F. A nouveau, la BD n'est pas corrompue ... à moins que la première transaction ne se serve de ce total erroné lors d'une mise à jour.

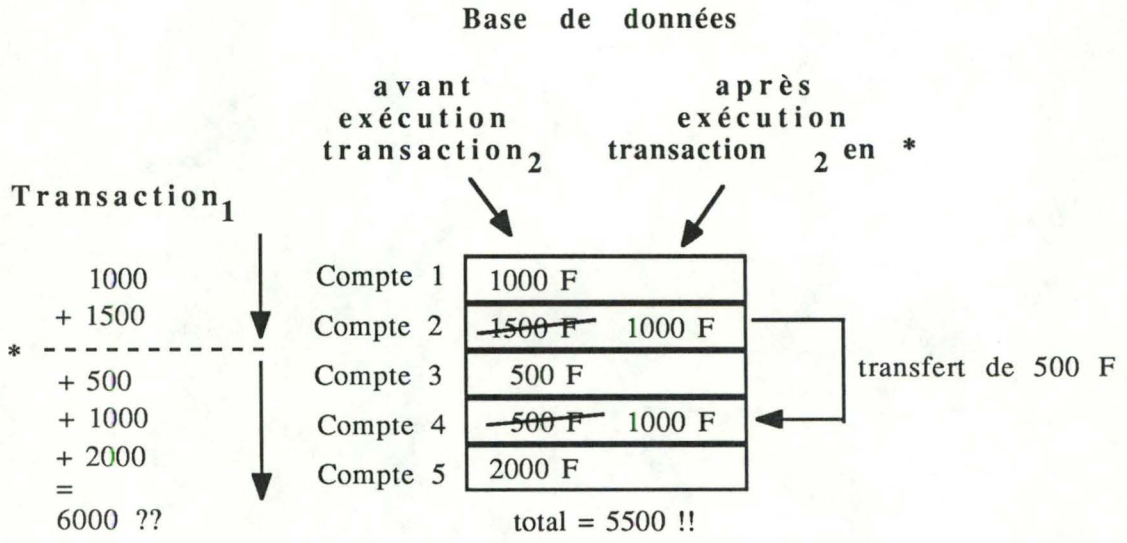


Figure 3 : Incohérence statistique

3. Acceptabilité des solutions générales pour un O.S.

Si l'on excepte la solution brutale de pure séquentialité des transactions, il existe des techniques qui ont été développées pour un usage dans les systèmes d'exploitation. Pourquoi alors ne pas essayer de récupérer ces méthodes pour les adapter aux BD ? Pour chacune de celles présentées juste ci-après (classification issue de [BAYE76]), nous tenterons de montrer en quoi elles sont difficilement applicables aux BD.

3.1. Sérialisation des processus

La sérialisation des processus consiste, comme nous le verrons dans le chapitre 2, à classer les processus susceptibles d'entrer en compétition pour une même ressource dans un ordre purement séquentiel.

La technique est difficilement applicable dans le domaine transactionnel car il faudrait pour cela connaître à l'avance les données auxquelles toutes les transactions vont accéder et que dès que deux d'entre elles sont en compétition pour la même ressource, les faire exécuter de façon purement séquentielle. La brutalité de la méthode en explique l'inadéquation pour les BD puisqu'on cherche justement à en augmenter la disponibilité; quelques techniques de contrôle de concurrence pour BD sont cependant inspirées de ce principe, mais nous verrons qu'elles ne se prêtent pas à des configurations générales sans certaines hypothèses.

3.2. Préemption des processus

La technique de préemption des processus accorde des priorités aux processus selon un schéma particulier puis, lorsque ces processus se trouvent en situation de bloquage mutuel pour une ressource, force le processus le moins prioritaire à abandonner la ressource qu'il détient, s'il en détient une, de façon à pouvoir l'accorder à l'autre. Cette technique sera souvent employée pour le contrôle de concurrence appliqué aux BD mais nous verrons qu'étant donné le grand nombre de ressources impliquées - toute la BD - elle risque devenir lourde à gérer.

3.3. Ordre préalable sur les ressources

Imposer un ordre préalable sur les ressources revient à les numéroter et à exiger que l'accès à chacune d'elles se fasse dans un ordre purement séquentiel en commençant par la première. Dans de nombreuses situations, imposer un ordre si strict s'avérera souvent excessif alors qu'un ordre hiérarchique aurait pu suffire.

De plus, dans les BD, il est rare que l'on puisse ordonner les objets de cette façon puisque les demandes à effectuer seront souvent dépendantes du contenu des données déjà lues.

3.4. Revendication des ressources nécessaires

Un processus auquel on impose la revendication des ressources nécessaires doit faire connaître, avant son exécution, la liste des ressources dont il aura besoin. Il ne pourra donc démarrer que lorsque toutes les ressources requises lui seront accordées. Cette technique propre aux ressources non-partageables s'annonce, sinon impossible, du moins lourde à

implémenter et à utiliser dans un environnement d'accès transactionnels, bien que certains algorithmes que nous présenterons s'en inspirent.

Nous voyons donc que les accès à une BD sont plus particuliers que les accès aux ressources d'un système d'exploitation et qu'ils nécessiteront donc des méthodes plus spécifiques. Cette spécificité est aussi remise en cause par les exigences d'intégrité de la BD qui diffèrent de celles d'un système d'exploitation pour les raisons suivantes :

◊ la longévité : même si les erreurs sont rares, suite à la longue durée de vie de la BD, celle-ci risque de souffrir d'une contamination et dégradation de la qualité de la BD;

◊ la répétabilité limitée : même si certaines erreurs sont décelées immédiatement, il peut être impossible de corriger la situation à cause de contraintes de temps ou d'indisponibilité de l'état correct avant erreur soit du système soit des données;

◊ accès à des utilisateurs multiples : le nombre et la variabilité des utilisateurs exige une gestion plus attentive du système où la source et la prolifération des erreurs sont parfois difficiles à assurer.

Nous tentons, dans la suite de ce travail, de présenter un synthèse des techniques de contrôle de concurrence spécifiques aux BD.

Chapitre 1

Le modèle de travail

Nous présentons à la figure 1.1 le modèle du SGBD dont nous nous servirons dans la suite de ce travail. Ce modèle est une variante de celui proposé dans [BERN87]. Nous expliquons un peu plus bas les rôles que nous attribuons aux différents composants que nous avons distingués.

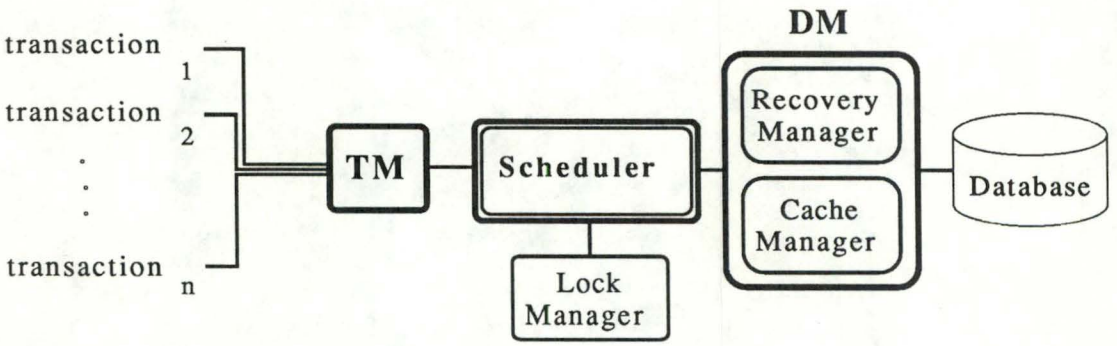


Figure 1.1 : Modèle de SGBD centralisée

Avant de donner lieu à l'implémentation finale du SGBD, ce modèle subira de nombreuses transformations - souvent pour des impératifs de performance - et il sera parfois bien difficile pour le lecteur de retrouver dans cette implémentation les traces du modèle logique dont on était parti. Des modules à rôles bien distincts au départ peuvent devenir fortement liés par la suite. Nous le mettons en garde afin qu'il ne confonde les dénominations de notre abstraction et les dénominations d'une quelconque réalisation physique.

Le gérant des transactions **TM** (pour Transaction Manager) constitue une interface obligatoire entre les transactions des différents utilisateurs et le reste du SGBD.

Le TM reçoit les demandes d'opérations sur la BD issues des transactions. Ces transactions peuvent être des requêtes interactives exprimées dans un langage d'interrogation de BD ou des parties de programmes d'application écrits dans un langage d'usage général.

Elles peuvent accéder à la BD moyennant quatre opérateurs que le TM lui reconnait :

- **TR_BEGIN** : définir le début de la transaction;
- **TR_END** : définir la fin de la transaction. Nous distinguerons si nécessaire la fin normale (TR_COMMIT) de la fin anormale (TR_ABORT);
- **TR_READ** (X) : lire la donnée désignée par X;
- **TR_WRITE** (X, VAL) : écrire la valeur VAL dans la donnée désignée par X.

*Nous n'incluons pas certaines opérations telles TR_DE*LETE (X) et TR_INSERT (X, VAL). *Nous souhaitons par là simplement simplifier les exemples qui suivront. Nous expliquerons au moment opportun comment cet écart n'affecte en rien la validité de nos dires.*

Le TM réalise quelques opérations préparatoires[1] sur ces demandes avant de les transmettre au scheduler. Ces opérations étant négligeables dans le cadre de notre travail, nous commettrons un abus de langage en disant que le scheduler reçoit ses opérations directement des transactions, alors qu'en réalité elles proviennent du TM.

Le **scheduler** est la partie qui nous intéresse tout spécialement. Il renferme en effet les algorithmes de contrôle de concurrence.

Lorsqu'il recevra par exemple une demande de lecture ou d'écriture d'une transaction, en fonction des circonstances, il pourra décider

◊ soit de transmettre la demande au DM,

◊ soit de la retarder momentanément,

◊ soit de la refuser.

C'est par ces trois moyens que le scheduler essaiera d'assurer les exécutions (parallèles) correctes des transactions.

Nous avons mis à sa disposition un module de gestion des verrous qui sera responsable de la pose et de la libération des verrous sur base des requêtes formulées par le scheduler. Pour une demande d'accès à une donnée, nous supposerons qu'a d'abord lieu une demande de verrou; si le verrou est accordé, l'ordre d'exécuter l'opération est transmis au DM, puis, le verrou libéré. Ce module ne sera évidemment nécessaire que pour les méthodes fonctionnant selon le principe du verrouillage.

Le gérant des données **DM**, quant à lui, effectue les accès physiques aux données requises par le scheduler. Il comporte une partie destinée à la gestion de la mémoire cache

[1] Telles que, par exemple, ajouter à l'opération demandée l'identifiant de la transaction.

(portion de la BD en mémoire volatile) et une partie destinée à la gestion des recouvrements. Cette dernière, responsable du maintien de la BD dans un état cohérent après un COMMIT, ABORT ou événement anormal, a été synthétisée dans [CHAN87].

Notons qu'aucun de ces modules n'est capable de prévoir ce qu'un autre lui demandera de faire. Chacun reçoit les demandes une à une et est libre de les traiter dans l'ordre qu'il souhaite. Si l'un des modules veut s'assurer que l'ordre de transmission des opérations est conservé, il doit s'en assurer lui-même. Il existe toutefois une alternative qui permet de ne pas imposer cette obligation aux différents modules; il s'agit de la file d'attente. Par défaut, nous supposerons que l'ordre n'est pas respecté, mais lorsque certaines techniques nécessiteront l'hypothèse de la file d'attente, nous le signalerons.

Chapitre 2

La sérialisabilité

Lors de l'exposé des problèmes dus aux accès concurrentiels à une BD partagée, nous avons fait remarquer que les contraintes d'intégrité de la BD pouvaient être violées de par le fait que, dans un environnement multi-utilisateurs, certaines actions d'une transaction seraient très probablement interrompues pour permettre à celles d'une autre de s'exécuter avant de reprendre la transaction en cours. Ces problèmes pouvaient très bien surgir malgré un système et des programmes sans faute.

Une manière de résoudre ces interférences est de toujours terminer l'exécution d'une transaction avant d'en commencer une autre. Ainsi, l'**exécution** d'une série de transactions sera dite **sériale** si, pour chaque paire de ces transactions, toutes les actions de l'une sont exécutées avant n'importe laquelle de l'autre.

Par hypothèse, une exécution sériale est correcte puisque, si la BD est cohérente au départ, les transactions étant exécutées atomiquement et séquentiellement (sans aucune interférence possible donc), elles héritent de leur prédecesseur une BD cohérente et la laissent dans un état cohérent pour leur successeur.

Toutefois, nous rappelons que cette solution, extrêmement simple, exclut tout parallélisme d'accès à la BD. Aussi, nous allons étendre cette classe d'exécutions admissibles de façon à accepter également les exécutions qui ont le même effet qu'une exécution sériale. Seront donc admises les exécutions **sérialisables**, c'est-à-dire celles qui ont le même effet sur la BD (et les mêmes résultats en sortie) qu'une des exécutions sériales possibles. Nous pouvons étendre cette classe en vertu du fait que, puisque les exécutions sériales sont correctes, et puisque chaque exécution sérialisable a le même effet sur la BD qu'une exécution sériale, alors les exécutions sérialisables sont correctes aussi.

Ainsi par exemple, les exécutions représentées aux figures 1 et 3 (mise à jour perdue et incohérence statistique) de l'introduction à cette première partie ne sont pas sérialisables. En effet, chacune d'elles laisse la BD dans un état différent de celui qui aurait été obtenu par une exécution sériale (c'est-à-dire purement séquentielle).

Remarque :

Etant donné une série de transactions, toutes les exécutions sériales n'auront pas nécessairement le même effet sur la BD. Prenons par exemple, un compte bancaire à

solde nul, et deux transactions : la première a pour but de créditer ce compte de 1000 F, la seconde de le débiter de 500 F. Ainsi exécutées séquentiellement, le solde serait de 500 F. Mais, en exécutant d'abord la seconde transaction, pour peu que la banque refuse au client de tomber en négatif, le solde serait de 1000 F.

Nous insistons sur le fait qu'une exécution est sérialisable si elle produit le même effet qu'UNE exécution sériale, peu importe laquelle. Si l'utilisateur préfère tel ou tel ordre d'exécution sériale des transactions, c'est à lui que cela incombera, et non au TM.

Dans la littérature, un séquencement particulier des "actions" - nous expliquerons desquelles un peu plus loin - d'un ensemble de transactions sera appelé un **historique** (ou "schedule" en anglais). Un historique dans lequel une transaction ne verrouille pas une entité déjà verrouillée par une autre transaction sera qualifié de **légal**. Tout naturellement, un historique donnant à chaque transaction une vue cohérente de la BD sera dit **cohérent**.

Ci–dessus, nous avons mis le terme "actions" entre guillemets sans le définir précisément. Nous avons voulu laisser à ce terme toute sa généralité : il englobe en tous cas les actions telles que TR_BEGIN, TR_END1, TR_READ et TR_WRITE; mais il peut aussi englober les actions telles que la pose (la libération) de verrous par exemple. Ces dernières sont particulières dans la mesure où le programmeur d'application ne les utilisera pas toujours : certains SGBD gèrent eux-même le (dé)verrouillage. Qu'elles soient du ressort ou non de l'utilisateur, les actions de verrouillage seront explicites dans la suite de notre travail.

La sérialisabilité a fait l'objet d'une théorie mathématique très complète qui permet de prouver si oui ou non un algorithme de contrôle de concurrence fonctionne correctement. Cette théorie a été dûment exposée, entre autres par Eswaran [ESWA76] et Bernstein [BERN79, BERN87]. Elle permet justement de dégager une condition suffisante pour qu'un historique soit cohérent et qui est que l'exécution de cet historique ait le même effet sur la BD qu'une exécution séquentielle des transactions prises en compte.

Un des outils de cette théorie porte le nom de "**graphe de sérialisation**". C'est son analyse qui permet par exemple de déterminer si tel ou tel historique est sérialisable. Nous ne définirons cette notion qu'au moment où nous en aurons besoin car les notions que nous avons développées jusqu'à présent ne suffisent pas encore.

---

[1] TR_END vaut tant pour le TR_COMMIT que pour le TR_ABORT. Nous utiliserons explicitement l'un ou l'autre de ces deux derniers si nécessaire. Notons que le TR_END est censé libérer tous les verrous que la transaction aurait posés mais qu'elle n'aurait pas encore explicitement relâchés.

Chapitre 3

Techniques de verrouillage

3.1. Principe de base

La technique du verrouillage est sans doute la méthode la plus simple quant aux concepts utilisés. Son **principe** de base est le suivant : lorsqu'une transaction veut accéder à une donnée, elle pose sur celle-ci un verrou qu'elle garde jusqu'à la fin de la transaction. Durant toute sa durée d'exécution, elle est protégée des interférences d'autres transactions qui voudraient accéder à cette même partie de la BD, puisque le scheduler leur signale la présence de ce verrou. La transaction qui a posé le verrou en premier lieu devient en quelque sorte propriétaire de l'objet. Les autres transactions désirant avoir accès au même objet resteront bloquées jusqu'à ce que la première libère celui-ci.

En principe, une seule transaction à la fois peut donc se voir garantir un verrou sur un même objet (en principe, car dans la suite de notre exposé, nous affinerons les types de verrous utilisés).

Ce protocole présente déjà l'avantage d'aider à la résolution des problèmes cités dans notre introduction à cette partie, à savoir ceux de la mise à jour perdue, de l'incohérence statistique et de la donnée fantôme (fig. 1, 2 et 3). Il permet d'assurer la sérialisabilité des exécutions, autrement dit des exécutions concurrentes correctes. Il présente cependant l'inconvénient majeur de réduire parfois inutilement les possibilités de parallélisme entre transactions. C'est pourquoi on trouvera une forme plus raffinée de cette technique, celle du verrouillage à deux phases auquel le point 3.2 est consacré.

Remarque :

La responsabilité du verrouillage peut soit être laissée aux soins de l'utilisateur soit être déléguée au système. La première solution offre, théoriquement, l'avantage de verrous plus rares de par la connaissance par l'utilisateur de l'usage futur des données. D'un autre côté, le verrouillage laissé aux soins du programmeur risque d'introduire une programmation difficile et peut-être peu fiable, outre le fait qu'il risque d'y avoir des abus. De là, la seconde approche qui est de charger le système de la gestion automatique du verrouillage, quitte à fournir à l'utilisateur des alternatives en vue d'une éventuelle optimisation.

3.2. Le verrouillage à 2 phases : 2PL

Le verrouillage à deux phases, **2PL** (pour 2 Phase Locking), est sans aucun doute la technique la plus populaire à l'heure actuelle. Si ce n'est pas dans sa version originale, c'est sous une forme dérivée. C'est pourquoi nous mettons le 2PL et ses variantes aux premiers rangs des techniques de contrôle de concurrence.

Pour la présentation du 2PL, nous utilisons une notation tirée des travaux de Bernstein et Goodman [BERN81], [BERN87].

Nous distinguons deux catégories de verrou : **exclusif** et **partagé**. Un verrou exclusif donne, par définition, un accès exclusif à la transaction qui le détient, alors qu'un verrou partagé autorise, sous certaines conditions, plusieurs transactions à y accéder. On posera typiquement un verrou exclusif sur une donnée qu'on souhaite modifier et un verrou partagé sur une donnée qu'on ne veut que lire.

Nous dirons que deux verrous quelconques sont en **conflit** si, à la fois,

◊ ils portent sur la même donnée,

◊ sont issus de deux transactions différentes,

◊ et si l'un au moins d'entre eux est un verrou de type exclusif.

Le tableau 3.1 ci-dessous reprend tous les cas de conflit possibles entre les deux types de verrous définis :

Tableau 3.1 : Conflits possibles entre verrous exclusif et partagé en 2PL

Compatibilité ?		Donnée verrouillée en Lecture	Donnée verrouillée en Ecriture
Donnée demandée en Lecture		OUI	NON
Donnée demandée en Ecriture		NON	NON

Notation :

Nous utiliserons RL[X] pour désigner un verrou de lecture sur la donnée X, et WL[X] pour un verrou en écriture sur X (RL pour ReadLock, WL pour WriteLock). Nous utiliserons donc $RL_i[X]$ (ou $WL_i[X]$) pour signifier qu'une transaction particulière T_i a obtenu un verrou en lecture (ou écriture) sur X. Nous utiliserons les lettres O, P ou Q pour dénoter des opérations quelconques (de lecture ou écriture), et $OL_i[X]$ pour le

verrou de type O obtenu par T_i sur X. La même notation sera aussi utilisée pour désigner les opérations qui posent ou obtiennent les verrous. Le contexte sera toujours assez clair afin de distinguer les verrous des opérations elles-mêmes.

D'une façon analogue, $RU_i[X]$ et $WU_i[X]$ désigneront respectivement les opérations de libération des verrous en lecture et écriture sur X pour une transaction T_i (RU pour ReadUnlock et WU pour WriteUnlock).

La technique du 2PL tient ce nom du fait que chaque transaction peut être découpée en deux phases : une phase de croissance et une phase de récession. Pendant la première, la transaction acquiert tous les verrous dont elle aura l'usage, verrous qu'elle libère pendant la seconde phase. Plus précisément, voici les **3 règles** qu'un scheduler doit respecter pour être considéré comme **2PL** - version d'origine - ([BERN87]) :

1° lorsqu'il reçoit une opération $P_i[X]$ du TM, le scheduler teste si le verrou $PL_i[X]$ est en conflit avec un quelconque verrou $QL_j[X]$ déjà posé. Si oui, il retarde $P_i[X]$ (forçant ainsi la transaction T_i à attendre jusqu'à ce qu'il puisse obtenir le verrou). Sinon, le scheduler pose le verrou $PL_i[X]$ et envoie la demande d'opération $P_i[X]$ au DM;

2° une fois que le verrou $PL_i[X]$ a été posé pour T_i, le scheduler ne peut libérer ce verrou qu'après avoir reçu confirmation du DM que celui-ci a réalisé l'opération correspondante $P_i[X]$;

3° une fois que le scheduler a relâché le verrou pour une transaction, il ne peut plus obtenir aucun autre verrou, sur quelque donnée que ce soit pour cette même transaction.

La première règle empêche que deux transactions n'accèdent à une donnée en mode de conflit. Les opérations en conflit sont donc servies dans l'ordre d'obtention des verrous.

La deuxième règle assure que le DM exécute bien les opérations dans l'ordre d'envoi par le scheduler. Il s'agit d'éviter que le TM, libérant un verrou ($RL_i[X]$ par exemple) de T_i avant que le DM n'ait confirmé l'opération correspondante (soit $R_i[X]$), n'accorde à une autre transaction T_j un verrou (en conflit auparavant) sur X (soit $WL_j[X]$), et envoie $W_j[X]$ au DM. Il n'y aurait alors pas de garantie que le DM exécute le $R_i[X]$ avant le $W_j[X]$. Ce cas de figure est intéressant dans la mesure où il permet de se rendre compte que libérer un verrou le plus tôt possible n'est pas toujours une solution intéressante.

La dernière règle, dite **règle des deux phases**, a un but moins évident. C'est elle qui a donné le nom à cette technique, ainsi que nous l'avions introduit plus haut. Afin de l'esquisser, nous pourrions dire qu'elle vise à garantir que toutes les paires d'opérations en conflit de deux transactions sont servies dans le même ordre.

L'exemple suivant permet de mieux comprendre ce que le non respect de la règle des deux phases implique :

soit deux transactions T_1 et T_2 définies comme suit :

T_1 :	TR_BEGIN_1	T_2 :	TR_BEGIN_2
	$TR\_READ_1(X)$		$TR\_WRITE_2(X, V_{21})$
	$TR\_WRITE_1(Y, V_1)$		$TR\_WRITE_2(Y, V_{22})$
	TR_COMMIT_1		TR_COMMIT_2

Si on considère l'exécution concurrente de ces deux transactions, alors on peut envisager l'historique suivant (de gauche à droite et de haut en bas dans le temps) :

$RL_1[X]$ $R_1[X]$ $RU_1[X]$

$WL_2[X]$ $W_2[X]$ $WL_2[Y]$ $W_2[Y]$ $WU_2[X]$ $WU_2[Y]$ TR_COMMIT_2

$WL_1[Y]$ $W_1[Y]$ $WU_1[Y]$ TR_COMMIT_1

Cet historique n'est cependant pas sérialisable (un exposé plus complet de la sérialisabilité et de son application à cet exemple est donné dans [BERN87]). Le problème provient du fait que la transaction T1 a relâché un verrou et ensuite posé un autre verrou, enfreignant ainsi la règle des deux phases. Si T1 avait respecté cette règle, T2 ne serait pas venue s'intercaler entre le RU1[X] et le WL1[Y], ainsi que le suggère l'encadré ci-dessus. T2 a en effet eu le temps de modifier et X et Y, sans que T1 n'en ait eu conscience; si les actions de T1 sur Y sont par exemple conditionnées par ce que R1[X] a pu fournir, alors elles seront sans doute erronées car X et Y ont tous deux été modifiés depuis.

Voici un exemple détaillé d'historique qu'un scheduler 2PL aurait pu et dû produire :

- ◊ *le scheduler reçoit $R_1[X]$, pose le verrou $RL_1[X]$ et soumet $R_1[X]$ au DM, qui confirme ensuite le traitement de $R_1[X]$;*
- ◊ *le scheduler reçoit $W_2[X]$, mais ne peut poser $WL_2[X]$ qui est en conflit avec $RL_1[X]$. Il met donc l'exécution de $W_2[X]$ en file d'attente;*
- ◊ *le scheduler reçoit $W_1[Y]$, pose le verrou $WL_1[Y]$ et soumet $W_1[Y]$ au DM, qui confirme ensuite le traitement de $W_1[Y]$;*

◊ *le scheduler recoit TR_COMMIT_1, signalant que T_1 est terminée. Il l'envoie au DM et, après confirmation du traitement par celui-ci, libère $RL_1[X]$ et $WL_1[Y]$* [1];

◊ *le scheduler peut maintenant poser le verrou $WL_2[X]$ de la requête $W_2[X]$ qui avait été retardée, requête qu'il envoie au DM et pour laquelle il attend confirmation;*

◊ *le scheduler reçoit $W_2[Y]$, pose le verrou $WL_2[Y]$, demande au DM de traiter $W_2[Y]$ et attend confirmation;*

◊ *T_2 se termine grâce au TR_COMMIT_2. Le scheduler le signale au DM et après traitement et confirmation de celui-ci, libère les verrous $WL_2[X]$ et $WL_2[Y]$.*

Remarque :

Jusqu'ici, ainsi que nous l'avions déjà annoncé lors de l'introduction de cette première partie, nous avons expressément omis d'envisager les cas du TR_DELETE (X) et du TR_INSERT (X, VAL).

*Le lecteur comprendra aisément que pour exécuter un **TR_DELETE**, le scheduler devra d'abord verrouiller l'objet X en mode exclusif puis, après obtention du verrou, demander sa destruction. Ceci ne pose guère de difficultés et s'apparente fortement à un TR_WRITE.*

*Le **TR_INSERT** quant à lui est un peu plus délicat. En effet, il n'est pas question de verrouiller un objet qui n'existe pas encore. Il faudra donc que le verrouillage (exclusif) de X soit immédiatement demandé par le scheduler pour la transaction émettant cette requête d'insertion. Cette demande peut être satisfaite immédiatement puisque X n'existait pas encore et qu'un conflit n'est donc pas possible. Il faut toutefois que ce verrouillage ait directement lieu après la création de X. On peut aisément imaginer ce qui se passerait si la transaction T1 était interrompue entre la création de X et son verrouillage par T2 qui accèderait justement à ce X. A l'exception de cette particularité, le TR_INSERT peut lui aussi être assimilé à un TR_WRITE.*

Pour prouver qu'un scheduler est correct (celui-ci en particulier), il faut prouver que tous les historiques qu'il peut produire sont sérialisables. C'est ce que font Bernstein, Hadzilacos et Goodman dans [BERN87] à l'aide du théorème de la sérialisabilité. Le protocole 2PL produit donc bien des historiques sérialisables. Malheureusement, cette technique fait partie des techniques avec verrouillage, et celles-ci sont sujettes à un inconvénient majeur : les interblocages (ou "deadlocks" en anglais, "verrouillage mortel"). C'est ce que nous développons dans le point suivant.

---

[1] La règle des deux phases est donc respectée puisque $R_1[X]$ et $W_1[Y]$ ont déjà été traitées et que T_1 ne'exigera plus aucun verrou.

3.3. Variantes du 2PL

Nous avions déjà fait remarquer plus haut que si le scheduler n'envoyait pas directement la requête de la transaction au DM, alors il pouvait soit retarder la requête (en la mettant dans une file d'attente, par exemple), soit la rejeter.

Ce choix permet de distinguer deux types de schedulers : les schedulers conservateurs dans le premier cas, les schedulers agressifs dans le second cas.

Le **scheduler conservateur** a pour politique de ne jamais avorter une transaction. Il n'exécutera les actions de celle-ci que s'il est certain de pouvoir les satisfaire jusqu'au bout. Le cas extrême étant la pure séquentialité des transactions. Ce type de scheduler essaiera donc d'anticiper le comportement futur des transactions afin de se préparer pour les demandes non encore reçues. Des informations valables pour atteindre cet objectif seront les données que la transaction demandera en lecture et en écriture.

Le scheduler agressif, quant à lui, évite de retarder les opérations : il essaie de les exécuter immédiatement, quitte à devoir faire marche arrière plus tard s'il n'arrive pas à obtenir une exécution sériale (en "défaisant" la transaction). Il travaille donc en instantané et n'a pas besoin d'essayer de circonscrire les données que la transaction manipulera.

Chaque politique a donné lieu à une variante du 2PL. Nous avons vu que le 2PL "pur" pouvait entraîner des interblocages, et qu'une fois ceux-ci détectés[1], il fallait avorter une transaction appelée "victime". Cette remarque explique la création de la première technique, le **2PL conservateur**. Nous verrons comment elle permet d'éviter les interblocages. Par contre la deuxième, qui porte le nom de **2PL strict** (ou agressif), ne cherche jamais à les écarter et nécessitera de ce fait une proportion plus importante d'avortements que le 2PL de base.

3.3.1. Le 2PL conservateur : la prédéclaration

Puisque cette version du 2PL est dite conservatrice, cela signifie qu'elle évite les interblocages. Pour y arriver, elle exige que la transaction ait obtenu tous ses verrous avant de procéder à une opération quelconque.

Nous ferons apparaître, lors de l'introduction consacrée aux interblocages, que ceux-ci proviennent d'un enchevêtrement des demandes de verrous mises en attente.

La solution à ce problème est celle de **la prédéclaration**. Elle consiste à demander que chaque transaction transmette au scheduler ses listes d'intention, à savoir l'ensemble

[1] La méthode de détection est sans doute la plus couramment utilisée.

des données qu'elle compte lire (appelé ReadSet[1]) et l'ensemble des données qu'elle compte modifier (appelé WriteSet).

Le scheduler essaiera d'obtenir tous les verrous nécessaires (spécifiés par les Read et Write sets) à une transaction T_i. Aucun de ceux-ci ne pourra donc être en conflit avec un quelconque verrou d'une autre transaction T_j.

S'il y arrive, il peut traiter les opérations provenant de T_i, c'est-à-dire les envoyer au DM. Ayant reçu confirmation de l'exécution de ces opérations par le DM, le scheduler peut enfin libérer tous les verrous de cette transaction.

Si par contre un de ces verrous est incompatible avec ceux déjà existants, le scheduler ne peut en accorder aucun pour cette transaction. Il doit au contraire mettre cette dernière dans une file d'attente. A chaque libération de verrou, il cherchera à satisfaire les clients de cette file.

Ce protocole exclut bien les deadlocks puisque soit une transaction T_i obtient tous ses verrous soit elle n'en obtient aucun. Il y a en quelque sorte atomicité de la pose des verrous d'une transaction.

Si T_i n'a pu poser aucun verrou, alors aucune transaction ne saurait se trouver en attente de T_i dans le WFG. T_i ne se trouvera donc jamais impliquée dans un cycle du WFG et, par conséquent, dans un interblocage.

L'avantage de l'absence d'interblocage est évidemment d'éviter par la même occasion des "défaire" de transactions. De façon plus nuancée, il y aura quand-même un "défaire" de transactions mais qui consistera simplement à libérer les verrous obtenus. Le gain dû à l'absence d'un quelconque mécanisme de détection et d'avortements est important et motive à lui seul l'existence de cette technique.

Elle apporte cependant un <u>double inconvénient</u> non négligeable :

- ◊ chaque transaction doit mentionner à l'avance l'ensemble des données qu'elle compte traiter. Pour des transactions courtes et dont le profil est bien connu à l'avance, ce n'est pas trop contraignant. Il n'en est pas de même pour des transactions plus longues qui doivent préciser toutes les données sur lesquelles elle voudra travailler. Ce sera souvent sinon impossible du moins fastidieux[2];

[1] "Set" est le mot anglais pour désigner un ensemble.

[2] Nous pouvons envisager que la transaction prédéclare explicitement ces ensembles, par l'intermédiaire de l'interface TR_BEGIN mofifié de façon adéquate, par exemple, mais si cette solution semble trop contraignante, nous pouvons aussi imaginer qu'un préprocesseur du langage de programmation se charge d'analyser les transactions et de créer implicitement leurs prédéclarations.

◊ le fait de devoir obtenir tous les verrous en une seule fois affaiblit d'autant plus fortement les possibilités de parallélisme que le nombre de données à verrouiller est important. La probabilité pour une transaction d'être mise en attente augmente en effet avec la probabilité d'un conflit, elle-même fonction croissante du nombre de verrous à obtenir en une seule fois. Le cas limite est celui où une transaction se trouve en situation de famine à cause de ses prédéclarations trop volumineuses (en taille relative de la BD).

3.3.2. Le 2PL strict

Suite aux remarques soulevées pour le 2PL conservateur, nous envisageons une deuxième technique basée sur le 2PL, mais cette fois-ci plus agressive, c'est-à-dire en évitant de retarder (parfois inutilement) l'exécution des transactions. Il s'agit du 2PL "strict". Ce qualificatif est utilisé avec la signification que la règle de sérialisabilité a encore été renforcée pour aussi assurer la possibilité de recouvrement.

Pour cette version stricte du 2PL, l'acquisition des verrous n'est soumise à aucune contrainte particulière, si ce n'est celle du 2PL lui-même[1]. La libération de ces mêmes verrous est par contre différente puisqu'on exige ici que tous soient libérés en même temps à la fin de la transaction, soit après que le DM ait confirmé le traitement du TR_END de la transaction.

La justification de cette technique est la suivante : lorsque le profil des transactions est imprévisible, le scheduler n'aura la certitude que la transaction a posé tous ses verrous et n'émettra plus d'opérations par la suite qu'au moment où il recevra le TR_END.

Remarque :

En réalité, on peut même raffiner cette règle en disant que :

◊ *les verrous de lecture peuvent être libérés dès que le scheduler reçoit le TR_END;*

◊ *les verrous d'écriture doivent être gardés jusqu'à ce que le scheduler ait reçu confirmation du DM que le TR_END a été exécuté.*

C'est assez logique puisque la lecture n'a pas affecté les données et que le TR_END n'apportera aucune modification à cet égard, ce qui n'est pas nécessairement vrai dans le cas d'une rédaction, selon le type de TR_END demandé (TR_COMMIT ou TR_ABORT).

[1] Voir section 3.2, le protocole 2PL.

Il s'agira bien sûr de trouver le compromis entre les deux approches, selon le type d'application par exemple. Ainsi, si la probabilité d'apparition de conflits est faible ou si les accès se font surtout en lecture, la politique agressive sera sans doute plus performante, puisque peu de transactions devront être avortées. Par contre, dans l'optique contraire, cette politique ne sera pas payante car le surcoût amené par toutes les transactions à défaire sera élevé. C'est sans doute ce qui explique le grand succès du 2PL strict parmi les techniques à base de 2PL.

3.3.3. Variante profitant des "shadow values"

Cette troisième variante du 2PL est une création de Bayer, Heller et Reiser. Ce ne sont pas les versions du 2PL qui manquent, mais nous reprenons celle-ci car elle nous paraît particulièrement intéressante. Elle profite en effet d'un mécanisme généralement implanté pour des besoins de recouvrement, afin d'augmenter les possibilités de parallélisme du SGBD.

Nous n'avons jusqu'à présent pas touché aux mécanismes appartenant au domaine du recouvrement. Nous le ferons pour ce point car il existe en matière de recouvrement un mécanisme très important qui est celui des "**shadow values** " (littéralement, les "valeurs-ombres").

Le principe des shadow values est destiné à faciliter le défaire ("undo" en anglais) des transactions. Nous avions jusqu'à présent supposé que les requêtes passées au DM provoquaient de sa part une action immédiate sur la BD réelle (pas nécessairement sur support stable).

Pour rappel, il existe de nombreuses raisons pour défaire une transaction. Citons par exemple la nécessité de réparer un interblocage, la violation de contraintes d'intégrité, une demande explicite d'un utilisateur qui a changé d'avis, etc...

Défaire une transaction revient à restaurer la BD dans l'état qui précédait cette transaction. Cette tâche n'est pas des plus faciles et toute une littérature lui est consacrée. Nous reprenons la technique des shadow values car son principe est simple, son efficacité grande et son application fréquente.

Le **principe** des shadow values est de ne pas répercuter les modifications demandées par une transaction immédiatement sur la BD réelle, mais sur une partie de la mémoire locale à chaque transaction. Cet espace privé est destiné à enregistrer toutes les modifications d'une transaction jusqu'à ce que celle-ci se termine. Si elle se termine par un COMMIT, ces modifications sont répércutées sur la BD réelle; si par contre elle se termine par un ABORT, alors cet espace privé est simplement détruit - la BD réelle n'ayant pas été modifiée depuis le début de la transaction, elle se retrouve bien dans l'état cohérent voulu.

Nous ne présentons ici que les caratéristiques essentielles de la méthode de Bayer et al., à savoir le protocole, les avantages et les inconvénients. Le lecteur pourra en trouver le détail et les preuves complètes dans [BAYE80].

Le protocole qui sert de base à celui de Bayer et al. est le 2PL strict. Ils ont choisi le 2PL pour son usage courant et cette version stricte pour la particularité qu'elle a d'éviter la cascade des recouvrements[1].

Ce qu'ils reprochent à ce protocole, c'est que lorsqu'une transaction T veut modifier une donnée X, elle doit acquérir un verrou exclusif sur X, effectuer les mises à jour et finalement libérer le verrou. Le verrou étant exclusif, aucune autre transaction ne peut accéder à X tant que T n'a pas fini.

L'idée, en utilisant les doubles des valeurs modifiées, est de toujours offrir aux transactions lectrices la satisfaction de pouvoir lire au moins une des deux valeurs d'une donnée (soit l'ancienne, soit celle qui est en cours de modification).

La solution triviale de toujours offrir l'accès à la valeur la plus récente n'est cependant pas admissible pour des raisons de cohérence. Considérons en effet deux transactions, T_1 et T_2 définies comme suit :

T_1 : TR_BEGIN_1	T_2 : TR_BEGIN_2
$TR\_READ_1(X)$	
	$TR\_WRITE_2(X, V_{21})$
	$TR\_WRITE_2(Y, V_{22})$
$TR\_READ_1(Y)$	
TR_COMMIT_1	
	TR_COMMIT_2

Cette exécution n'est pas sérialisable (et doit donc être refusée) car elle revient à donner à T_1 la valeur de X avant exécution de T_2 mais la valeur de Y après exécution de T_2.

Il faut donc distinguer **trois états** différents d'un objet O :

E_1 : O n'a qu'une seule valeur; toutes les transactions lisent cette valeur;

E_2 : O n'a qu'une seule valeur; toutes les transactions lisent cette valeur sauf une qui prépare une nouvelle valeur de O;

E_3 : O a deux valeurs, O_ancienne et O_nouvelle; celle qui sera donnée en lecture aux autres transactions dépendra de la situation[2].

---

[1] En général, une transaction qui doit être défaite entraîne en effet la nécessité de défaire toutes celles qui en "dépendent", c'est-à-dire toutes celles qui se sont, d'une manière ou d'une autre, basées sur des inputs (données en entrée) provenant de la première.

[2] Une transaction ne pourra modifier O que s'il n'existe qu'une seule valeur de cet objet.

Trois types de verrous seront aussi nécessaires :

r : verrou de lecture car le scheduler ne pourra pas toujours renvoyer la nouvelle valeur de O et devra donc savoir s'il reste encore des lecteurs de O;

a : verrou d'analyse, utilisé par les rédacteurs pour annoncer qu'une nouvelle valeur de O se prépare;

c : verrou de COMMIT, utilisé par les rédacteurs pour faire savoir que la nouvelle valeur de O est maintenant disponible.

Remarques :

Un objet O sur lequel ne seront posés que des verrous "r" seront dans l'état E_1.
Si un verrou "a" et plusieurs verrous "r" sont posés sur O, alors O sera dans l'état E_2.
Sinon, il sera dans l'état E_3.
Le verrou "c" sur O pourra être relâché lorsqu'il n'y aura plus de lecteurs de O.

Le tableau des compatibilités sera alors le suivant :

Tableau 3.2 : Tableau des compatibilités

Compatibilité ?		Donnée verrouillée en mode		
		r	a	c
Donnée demandée en mode	r	OUI	OUI	OUI
	a	OUI	NON	NON
	c	OUI	NON	NON

Une transaction rédactrice T voulant modifier un objet O devra alors respecter le protocole suivant :

◊ poser un verrou "a" sur O;

◊ modifier O;

◊ convertir "a" en "c"; T est maintenant certaine de ne plus être défaite;

◊ libérer le verrou "c" lorsqu'il n'y aura plus de lecteurs de O.

Le protocole pour une transaction lectrice se résume grossièrement de la façon suivante :

a) dans le cas trivial d'une valeur unique, la lui accorder;

b) si la donnée est verrouillée en mode d'analyse, lui accorder l'ancienne valeur de la donnée, et,

c) si elle est verrouillée en mode de commit, lui accorder la nouvelle valeur, à moins que cela ne provoque une incohérence. Cette dernière opération est la plus difficile et nécessite la gestion assez complexe d'un graphe de dépendances. Nous rappelons que ce protocole est détaillé dans [BAYE80].

Nous terminerons la présentation de cette méthode par une évaluation sommaire :

◊ elle garantit qu'un lecteur obtiendra toujours l'accès à une donnée;

◊ elle assure qu'un lecteur ne sera jamais défait même pour réparer une situation d'interblocage ou pour des raisons de cohérence,

bref, elle offre de meilleures conditions de parallélisme en utilisant de outils déjà présents pour des raisons de recouvrement, même si

◊ elle exige une gestion assez complexe de graphes;

◊ certains rédacteurs devront parfois être défaits pour des raisons de cohérence;

◊ dans certains cas, les lecteurs obtiendront des valeurs de données plus anciennes qu'ils n'auraient obtenues avec un autre protocole.

3.4. Conversion de verrous

Il existe des situations où une transaction détenant un type de verrou sur un objet souhaite changer ce type de verrou afin de pouvoir effectuer une autre opération dessus.

Afin d'illustrer les enjeux d'un tel mécanisme, construisons les deux exemples suivants :

(1) une transaction désire modifier tous les objets satisfaisant à tel ou tel critère parmi un ensemble de données;

(2) une transaction T souhaite lire une donnée X, effectuer de longs calculs sur tout un ensemble d'autres données et modifier X en conséquence.

Sans la possibilité de changement de verrou proposée ci-dessus, dans l'exemple (1), la transaction serait obligée

◊ soit de parcourir toutes les données de l'ensemble en verrouillant chacune d'elles en mode écriture. Cette solution, les verrous d'écriture entrant plus souvent en conflit que les verrous de lecture, risque de rendre la transaction fort exigeante[1] et sujette à de longues attentes;

◊ soit de les parcourir en mode lecture, puis, pour chaque donnée satisfaisant au critère, libérer le verrou de lecture et demander un nouveau verrou, d'écriture cette fois. Cette solution n'est pas possible telle quelle pour un protocole de verrouillage de type 2PL car, la libération d'un verrou déclenchant la phase de récession de la transaction, elle ne pourrait plus exécuter d'opérations ultérieurement.

Dans l'exemple (2), la transaction devrait verrouiller X en mode exclusif, lire toutes les autres données et effectuer ses calculs et enfin effectuer la mise à jour de X. Cette solution est mauvaise puisque pendant toute la durée des calculs, l'accès à X est interdit aux autres transactions alors que T n'effectue la mise à jour de X que pendant ses toutes dernières minutes d'exécution. De plus, si la "granularité"[2] du verrouillage est grossière, il serait souhaitable qu'après modification de X, la transaction permette à nouveau aux autres d'accéder à X et elle-même de continuer.

Nous constatons par là qu'il serait intéressant de pouvoir disposer d'une **conversion de verrous** afin d'offrir à une transaction la possibilité de changer le type d'un verrou en cours d'exécution, un peu comme si elle émettait une nouvelle requête mais sans l'obliger à se terminer.

Afin d'adapter un scheduler 2PL de manière à ce qu'il accepte la conversion de verrous, il faudrait rendre cette conversion "atomique"[3] et procéder comme suit :

◊ une conversion (incrémentale) de S->X peut être acceptée par le scheduler à condition qu'il s'assure que la donnée n'est plus verrouillée en quelque mode que ce soit par une quelconque transaction;

◊ une conversion (décrémentale) de X->S pourrait être acceptée a priori sans aucune vérification par le scheduler puisque par la définition-même du verrou exclusif, il est certain qu'aucune autre transaction ne peut détenir de verrou sur cet objet. Toutefois, si cette conversion était acceptée, on retrouverait le problème dit des "données fantômes" présenté lors de l'introduction à cette partie.

---

[1] Elle demande en effet plus de privilèges qu'elle n'en a besoin. Là où un verrou partagé suffirait, elle doit acquérir un verrou exclusif.

[2] Nous verrons dans un point spécialement consacré à la granularité (section 3.6), qu'il s'agit de la taille relative des verrous, peu importe celle de l'objet verrouillé. Elle varie typiquement de la BD complète à l'enregistrement.

[3] Cela signifie que la conversion ne doit pas donner lieu à une libération de verrou puis à une réacquisition de verrou, auquel cas elle contredirait la règle de formation des transactions 2PL (croissance/décroissance), mais doit être considérée comme une seule opération de transformation.

L'exemple suivant (fig. 3.1) permet de se rendre compte en quoi la conversion de verrous peut être dangereuse.

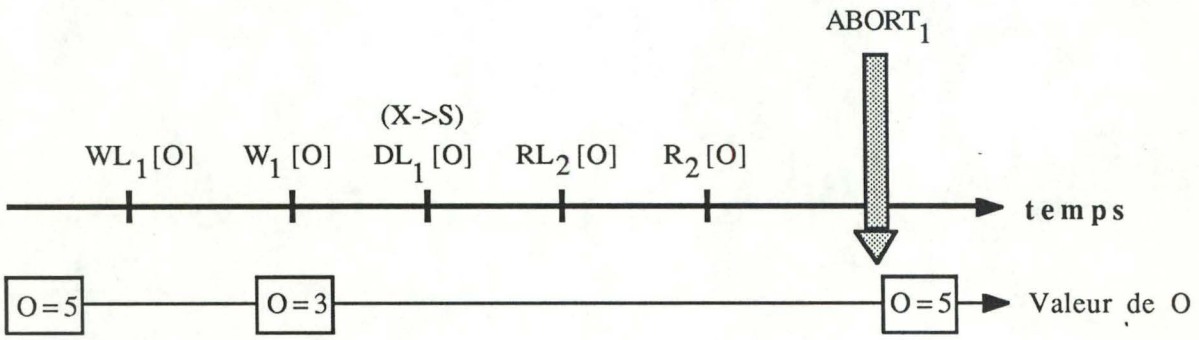


Figure 3.1 : Conversion X->S et donnée instable

Soit T1 une transaction qui, après avoir obtenu un verrou exclusif sur O (qui vaut 5), le modifie (en O=3) et demande une conversion de verrou de X->S (Decrement Lock) afin de permettre à d'autres transactions d'accéder à O. T2 peut maintenant verrouiller O en mode lecture et lire son contenu (O=3). Supposons que, pour une raison arbitraire, T1 doive être défaite. Par le principe de l'avortement, O est restauré dans son état "avant T1" et T2 a donc lu une valeur de O qui n'a jamais réellement existé (O=3 au lieu de O=5).

De là la déduction que le 2PL ne se prête pas tel quel à la conversion de verrous.

Outre la prudence avec laquelle certaines conversions doivent être exécutées, il convient de signaler qu'elles sont aussi sources d'interblocages ainsi que nous l'expliquerons plus loin dans le point consacré précisément aux interblocages.

La conversion de verrou est cependant suffisamment attrayante que pour avoir donné lieu à plusieurs protocoles qui la permettaient[1]. Mohan et al. en proposent quelques uns, de type non-2PL, qui offrent cette possibilité ([MOHA82], [MOHA85]).

[1] IMS d'IBM offre par exemple la conversion S->X.

3.5. Les problèmes liés au verrouillage

Nous avons vu comment de nombreux utilisateurs avaient la possibilité de revendiquer et de s'approprier certains éléments d'une BD afin de se protéger des interférences des autres pendant leur activité. Cette liberté peut toutefois engendrer des problèmes si elle reste incontrôlée. On en distingue généralement de deux types : les "deadlocks" et les "livelocks".

Le **deadlock**, encore appelé verrouillage mortel ou interblocage, est typiquement une situation où une transaction bloque une autre qui elle-même bloque la première. Cette boucle d'attente peut évidemment inclure de multiples transactions, mais au plus les transactions impliquées sont nombreuses, au moins une telle situation a de chances de se produire.

La difficulté en matière d'interblocage consiste sinon à l'empêcher du moins à le détecter de façon à pouvoir le réparer. Dans cette dernière hypothèse, il faudra donner la priorité à l'une des transactions mais avec des risques de violation de la cohérence de la BD.

Le **livelock**, encore appelé hibernation ou famine, est un cas de figure où une transaction est mise en attente infinie, parce que le système est trop occupé avec d'autres, mais où il n'y a malgré tout pas de deadlock.

Les deux types de problèmes empêchent l'utilisateur de voir s'accomplir son travail. Nous verrons cependant que, l'interblocage se prêtant bien à une analyse formelle, les SGBD pourront disposer d'outils de détection et de réparation sûrs. Par contre, pour ce qui est du phénomène de la famine, ce sera plus difficile. Elle est en effet souvent liée à la charge du système et n'est mise en évidence par aucun signe probatoire. Il vaudrait donc mieux implémenter des algorithmes qui permettent de les éviter.

La figure 3.2 permet de mieux visualiser le cheminement des opérations des transactions au travers des différents modules que nous avions discernés. Il devrait faciliter la compréhension des deux types de problèmes que nous développons ci-après.

Nous n'avons détaillé sur cette figure que le rôle du scheduler (celui du TM, limité d'ailleurs, et celui du DM ne sont pas en question dans ce travail). Les transactions sont émises par un certain nombre d'utilisateurs. Lorsqu'un de ceux-ci émet un TR_BEGIN, le scheduler s'apprête à recevoir les opérations constitutives de cette transaction. Chacune d'elles est mise dans la file d'attente à l'entrée du scheduler. Dès qu'il le peut, le scheduler la prend en charge.

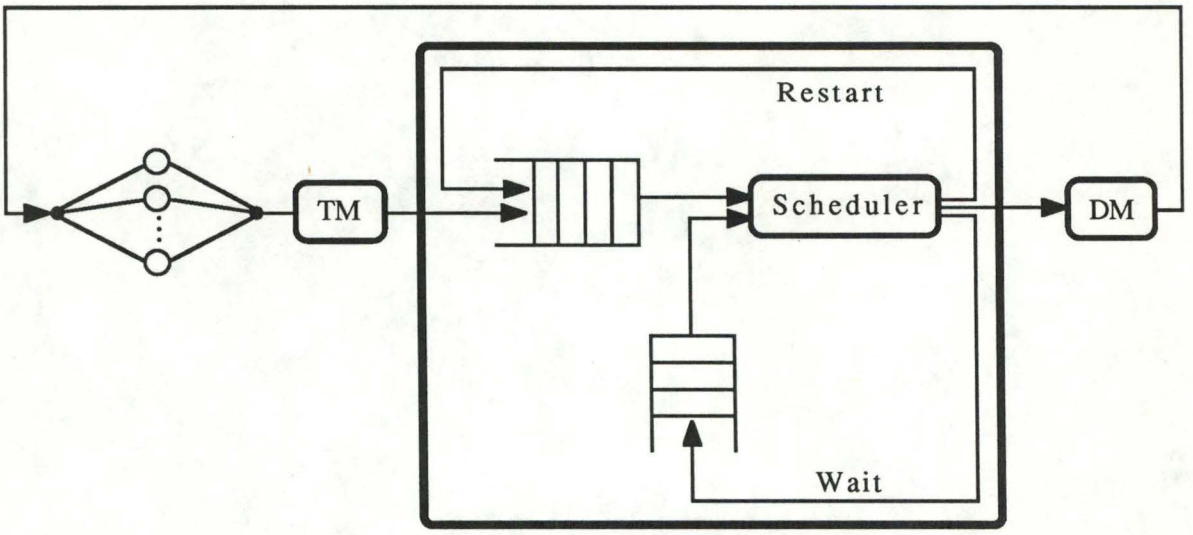


Figure 3.2 : Cheminement des opérations d'une transaction

En fonction de l'état du système ou du type d'opération demandée, cette demande se trouve dans une des trois possibilités suivantes :

◊ elle peut être satisfaite immédiatement, auquel cas le scheduler la transmet DM (avec dialogue préalable si nécessaire[1]);

◊ elle ne peut être satisfaite immédiatement; elle est alors mise dans la file d'attente des opérations bloquées;

◊ elle ne pourra jamais être satisfaite dans la configuration actuelle et doit de ce fait être redémarrée.

[1] Les échanges de messages entre modules ne sont pas repris sur ce modèle.

3.5.2. Les "deadlocks"

La première catégorie de problèmes résultant des techniques de verrouillage est celle des deadlocks, des interblocages. Nous rappelons qu'intuitivement, il s'agit d'une situation dans laquelle une transaction est bloquée par une ou plusieurs autres elles-même bloquées par la première. Il y aurait donc un cycle dans le schéma d'attente des transactions. C'est ce que nous essaierons de montrer à l'aide d'un exemple avant de formaliser le problème.

Prenons par exemple les deux transactions triviales définies ci-dessous :

T_1 : TR_BEGIN_1	T_2 : TR_BEGIN_2
$TR\_WRITE_1 (A, V_{11})$	$TR\_WRITE_2 (B, V_{21})$
$TR\_WRITE_1 (B, V_{12})$	$TR\_WRITE_2 (B, V_{22})$
TR_COMMIT_1	TR_COMMIT_2

Si elles sont exécutées concurremment, alors on peut imaginer que le scheduler respectera le début de scénario suivant :

$WL_1[A]\ W_1[A]\ WL_2[B]\ W_2[B]\ ...$

Dès à présent, l'interblocage existe. En effet, le scheduler que nous supposons 2PL version de base, ne peut effectuer ni WU1[A] ni WU2[B] puisque aucune des deux transactions n'est terminée. Il va donc devoir procéder aux demandes $TR\_WRITE_1 (B, V_{12})$ et $TR\_WRITE_2 (A, V_{22})$. L'ordre n'a pas d'importance pour cet exemple. Prenons qu'il commence par la demande de T_1. Le scheduler devra donc exécuter un $WL_1[B]$. Malheureusement, B est une donnée déjà verrouillée en mode exclusif par T_2. T_1 sera donc mise en attente. Le scheduler peut maintenant s'occuper de la demande de T_2, et envisager un $WL_2[A]$. A nouveau, cette action ne peut être réalisée car A est déjà bloquée par T_1. Les deux transactions se bloquent donc mutuellement.

L'exemple donné n'implique que deux transactions. Il peut paraître un peu trop trivial, mais c'est pourtant la situation la plus fréquente. Il serait bien sûr tout à fait possible de construire des cas semblables avec plusieurs transactions, mais la logique et la pratique montrent que la probabilité d'apparition de telles situations diminue avec le nombre de transactions qui devraient être bloquées.

L'exemple suivant montre comment un interblocage peut aussi être provoqué par la conversion de verrous (lorsque cette facilité est offerte).

Imaginons deux transactions T_1 et T_2 devant modifier un objet X satisfaisant à tel ou tel critère. Elles devront donc parcourir un ensemble de données et vérifier si elles correspondent au critère auquel cas elles effectueront la mise à jour. Afin de ne pas bloquer inutilement tout l'ensemble des données en mode exclusif, elles peuvent profiter

de la conversion de verrous en ne verrouillant les objets à lire qu'en mode partagé puis, lorsque le critère est satisfait, demander la conversion de verrou S->X avant d'effectuer ladite mise à jour. Si l'opération IL (Increment Lock) représente cette conversion de verrou, la figure 3.3 illustre bien nos propos :

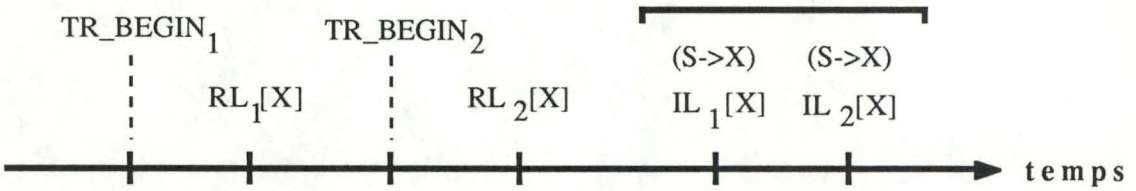


Figure 3.3 : Interblocage par conversion de verrou

Nous voyons que l'interblocage provient du fait que les deux transactions ont verrouillé le même objet en mode lecture puis, se rendant compte que le critère est satisfait, demandent toutes deux la conversion S->X. La conversion d'un verrou vers un mode exclusif ne pouvant se faire que si aucune autre transaction ne détient de verrou sur cet objet, T_1 et T_2 sont mises en attente l'une de l'autre.

Cette probabilité d'apparition d'un conflit varie aussi en fonction du nombre de transactions et de la taille de la BD en jeu. En fait, elle sera généralement fonction du nombre de transactions s'exécutant sur le système et fonction inverse de la taille de la BD. Elle variera aussi avec le profil des transactions : si elles sont principalement lectrices, les deadlocks se feront plus rares; mais si elles font de nombreuses mises à jour, alors ils seront sans doute beaucoup plus fréquents.

La littérature a répertorié **quatre conditions** nécessaires pour provoquer un interblocage :

◊ exclusion mutuelle : au moins une des ressources doit être détenue en mode exclusif;

◊ "détient et attend"[1] : il doit exister un processus détenant au moins une ressource et attendant de pouvoir acquérir des ressources supplémentaires détenues par d'autres processus;

◊ pas de "préemption" : signifie qu'un processus nouveau ne peut déposséder un processus plus ancien de ses ressources. Ce dernier doit donc libérer volontairement les ressources qu'il détient;

[1] "Hold and Wait" en anglais.

◊ attente circulaire : il doit exister un cycle dans l'ensemble des processus s'attendant l'un l'autre.

Notons que les quatre conditions doivent se présenter simultanément, mais que la dernière impliquera généralement les trois premières.

Une situation d'interblocage ne se réparera pas d'elle-même : s'il n'y a aucune intervention d'une instance quelconque, ces deux transactions demeureront en permanence dans cet état. Ce sera le rôle des techniques de détection. Une alternative serait d'éviter que l'interblocage ne se produise, et c'est là le rôle des techniques de prévention. Une troisième possibilité, celle du timeout, sera aussi présentée. Elle est un peu en marge des deux méthodes citées ci-dessus de par le fait qu'elle ne repose sur aucune certitude, mais bien sur une supposition - plus ou moins risquée - qu'il y a interblocage. Nous ne les développerons toutes trois qu'après avoir exposé les bases mathématiques qui serviront peu ou prou à les aider.

3.5.2.1. Le "timeout"

La stratégie du "timeout" (délai d'attente) est la première solution que nous présentons au problème de l'interblocage.

Comme nous l'avons déjà suggéré plus haut, elle ne repose sur aucune certitude de deadlock. En effet, si le scheduler constate qu'une transaction attend depuis déjà trop longtemps qu'un verrou soit libéré, alors il suppose qu'il y a peut-être un interblocage et sur ce décide d'avorter[2] cette transaction. Mais il s'agit là simplement d'une supposition d'interblocage. Le scheduler fait peut-être une erreur. Il se peut très bien que cette transaction attende un verrou posé par une autre transaction (non infinie) qui met longtemps à s'exécuter avant de se terminer.

Une telle erreur de jugement de la part du scheduler n'est pas très grave puisque la BD est restaurée dans un état cohérent et que la transaction peut être relancée. Elle pénalise toutefois de façon indue une transaction "innocente" quoique, du point de vue des performances, le taux de traitement des transactions[3] en ressort généralement amélioré [BERN87].

Afin d'éviter ce type d'erreur, on peut envisager d'augmenter le délai d'attente. En effet, au plus longtemps le scheduler attend avant d'avorter une transaction, au plus les chances d'avorter réellement bloquée augmentent. Toutefois, ce raisonnement se trouve face à un nouvel inconvénient qui est que si une transaction est vraiment en situation de

---

[2] Avorter une transaction, c'est défaire cette transaction (et donc annuler les effets qu'elle aurait pu avoir sur la BD).

[3] "Throughput" en anglais

deadlock, alors elle devra aussi attendre plus longtemps avant que le scheduler ne le remarque et s'en charge.

Il s'agit à nouveau d'un compromis : le délai d'attente devrait être suffisamment long que pour assurer d'avorter une transaction bel et bien bloquée; il devrait aussi être assez bref que pour éviter à une transaction en deadlock d'être remarquée rapidement.

La méthode du timeout est en quelque sorte aussi une technique de prévention puisque le système de réslution des interblocages n'est pas certain que ceux-ci existent. Il en soupçonne simplement l'existence et avorte les transactions en conséquence.

Cependant, même si cette technique fixe une limite supérieure au temps d'exécution d'une transaction, puisqu'une transaction bloquée pendant trop longtemps est avortée, elle avorte aussi de plus en plus de transactions lorsque la charge du système augmente, puisque les ressources se font relativement plus rares et les temps d'attente plus longs. A la limite, nous pourrions envisager que cette tendance ait un effet auto-régulateur sur le système; les temps de réponse devenant trop lents, les utilisateurs auraient en effet tendance à abandonner leur tâche présente, lorsque la nature de celle-ci le permet.

Nous devons donc admettre que ces systèmes seront généralement acceptables pour des systèmes légèrement chargés mais inappropriés pour des systèmes congestionnés.

3.5.2.2. Le graphe des mises en attente : WFG

Nous avons vu plus haut comment les mathématiques sont venues appuyer la sérialisabilité. Elles seront à nouveau d'une aide précieuse en ce qui concerne la résolution des interblocages. Puisque ceux-ci impliquaient la notion de cycle, il paraît tout naturel de s'intéresser à la théorie des graphes. Nous en expliquerons les notions essentielles afin de permettre au lecteur de comprendre les principes mis en oeuvre sans toutefois entrer dans des points trop spécifiques.

Le graphe des mises en attente ou **WFG** (Wait-For Graph) est construit de la façon suivante :

- ◊ les noeuds du graphe sont des transactions;
- ◊ les arcs du graphe sont orientés, indiquent une attente et sont manipulés par le scheduler[1];
- ◊ le scheduler trace un arc de Ti vers Tj lorsque la transaction Ti attend que Tj libère le verrou posé sur une donnée à laquelle elle veut aussi accéder;
- ◊ et le scheduler efface cet arc lorsque Tj relâche le verrou.

[1] C'est lui qui est censé résoudre les interblocages.

Pour les méthodes de détection, il y aura bien évidemment interblocage lorsque le WFG contiendra un cycle. Pour les méthodes de prévention, il faudra éviter que ce cycle ne se produise. Certains algorithmes utilisés seront donc très semblables dans les deux cas. La fréquence de vérification sera cependant différente selon l'algorithme choisi, comme nous le verrons pour chacun d'eux.

3.5.2.3. La détection

Puisqu'on parle de détection, c'est que l'interblocage a déjà eu lieu. Les algorithmes de détection seront donc chargés de le détecter et de le réparer selon l'esquisse suivante :

◊ choisir une victime;

◊ défaire[1] la victime;

◊ satisfaire une attente;

◊ refaire[2] la victime.

A lui seul, cet algorithme révèle les enjeux majeurs des méthodes de détection :

◊ Comment détecter l'interblocage ? Et à quels moments ?

◊ Quelle victime choisir ?

◊ Défaire la victime, ce qui ne devrait pas poser trop de problèmes puisque les techniques de *backup* existent normalement déjà à d'autres fins[3].

◊ Satisfaire une attente : toutes les transactions impliquées dans l'interblocage y trouvent leur compte. C'est bien sûr l'objectif des méthodes de détection.

◊ Refaire la victime : celle-ci a été défaite pour permettre aux autres de continuer. Rien ne devrait l'empêcher de se réexécuter, éventuellement à l'insu de l'utilisateur, si ce n'est un léger délai.

1° La fréquence de vérification

Nous avions déjà suggéré que la fréquence à laquelle se faisaient les recherches de cycles dans le WFG variaient selon les méthodes utilisées, détection ou prévention[4].

Le cas des méthodes de prévention sera développé dans un point spécialement consacré à celles-ci.

---

[1] C'est-à-dire avorter la transaction.

[2] C'est-à-dire relancer l'exécution de la transaction.

[3] Le recouvrement par exemple.

[4] Pour les méthodes préventives, il n'y aura pas à proprement parler de recherche de cycle, puisque par définition celui-ci est supposé inexistant, mais il y aura recherche de possibilité de cycle.

Le processus de détection peut être déclenché :

a) chaque fois qu'une demande de verrouillage d'une transaction est mise en attente,

b) périodiquement, et

c) jamais.

La première opportunité exige une gestion presque permanente du WFG. En effet, l'accord d'un verrou à une transaction ne provoquera jamais d'interblocage[1], ni la libération d'un verrou. On ne devrait donc jamais rechercher de cycles dans le WFG plus fréquemment que lorsqu'une attente se déclare. Cette vérification provoque cependant une forte surcharge de travail qui devient d'autant plus importante que le nombre d'utilisateurs du système augmente. De surcroît, cette vérification donnera très souvent des résultats négatifs[2].

Suite à cette dernière remarque, on peut se demander s'il est bien nécessaire de rechercher les cycles après chaque insertion d'un arc dans le graphe des mises en attente [BERN87]. On pourrait en effet n'envisager cette opération qu'après qu'un certain nombre d'arcs aient été insérés (les deadlocks ne disparaissant pas d'eux-même). Les coûts de détection en sont donc fortement diminués, mais par la même occasion, un interblocage peut rester pendant un long moment non détecté. De plus, à chaque vérification, il faudra retrouver TOUS les cycles, et pas seulement ceux provenant du dernier arc inséré.

La deuxième opportunité est de rechercher périodiquement les cycles dans le WFG. Elle aussi a été envisagée suite à la remarque développée ci-dessus. Il y a là à nouveau matière à compromis : allonger les périodes augmente la probabilité de vraiment trouver un deadlock (au moins) et diminue bien sûr le coût de gestion du WFG, mais risque de débloquer les transactions en situation de deadlock avec un certain retard. Il faudra trouver un équilibre entre le coût de la détection et le coût d'une (ou plusieurs) détection(s) tardive(s).

La dernière opportunité consiste à ne jamais rechercher les interblocages. C'est un peu comme si on les recherchait périodiquement mais avec une période très longue [GRAY79]. Cette solution n'est qu'un pis-aller. Elle compte sur une absence presque totale d'interblocages et sur un mécanisme de détection de processus "errants" (boucles infinies, etc...) présents dans la plupart des systèmes. Cette tâche n'est évidemment pas du ressort du scheduler.

---

[1] Puisque le WFG ne comporte que des arcs d'attente.

[2] A savoir : "Non, cette mise en attente n'amène pas de cycle dans le WFG".

2° *Choisir la victime et la défaire*

Une fois qu'un cycle a été détecté, il faut choisir la transaction - tout naturellement appelée "victime" - qui devra relâcher un verrou afin de permettre à une transaction bloquée d'être servie.

La victime ne sera pas nécessairement la transaction qui a réellement provoqué l'interblocage (ce qui ne serait d'ailleurs possible que dans le cas où la détection est réalisée chaque fois qu'une demande de transaction est mise en attente, cfr. supra). Nous choisirons en effet la victime de façon à minimiser le coût de réparation du ou des deadlocks.

En supposant qu'il n'y ait qu'une seule transaction à "tuer", à moins de la tirer au sort, plusieurs critères doivent être considérés lors de l'évaluation de ce coût :

- ◊ le montant d'effort qui a déjà été investi dans la transaction, puisque cet effort sera perdu si la transaction est avortée;
- ◊ le coût de l'avortement de cette transaction, qui dépend généralement du nombre de mises à jour qu'elle a déjà effectuées;
- ◊ la propension de la victime à se retrouver impliquée dans un interblocage après avoir été défaite. Il se peut en effet que dès que la victime est relancée, elle provoque un nouveau deadlock. Afin d'éviter cela, une transaction trop souvent avortée à cause d'un deadlock ne devrait plus être choisie comme victime, à moins qu'il ne soit impossible de faire autrement. Une manière de s'en sortir est par exemple de choisir la transaction la plus "jeune" comme victime;
- ◊ le montant d'effort qui sera nécessaire afin de terminer l'exécution de l'éventuelle victime. On veut éviter l'arrêt d'une transaction qui était bientôt arrivée à sa fin. Le scheduler devrait dans ce cas être capable de prédire le comportement futur de la transaction en cours;
- ◊ le nombre de cycles qui contiennent cette transaction. Une seule transaction pouvant en effet être la cause de plusieurs interblocages, il serait préférable de défaire celle qui ferait partie du plus grand nombre de cycles.

Il faudra aussi éviter qu'une transaction, par des avortements répétés, ne se retrouve dans la situation du livelock telle que nous l'avons présentée plus tôt.

Quant au fait d'avorter une transaction, nous rappelons qu'il implique de défaire tout ce qui a déjà été réalisé dans son cadre et de libérer tous les verrous qu'elle aurait acquise. Les ressources ainsi libérées - il ne s'agit pas seulement de morceaux de la BD, mais aussi d'autres ressources du système - deviennent bien sûr disponibles aux autres transactions.

La portée d'un "défaire" est donc plus large que celle de la BD uniquement, elle touche tout le système et c'est là une des responsabilités du module de recouvrement[1].

3° Satisfaire les attentes et refaire la victime

Dès que les victimes ont été défaites, leurs verrous sont libérés et le scheduler peut satisfaire la prochaine transaction dans la file d'attente. Ceci ne pose donc guère de problèmes. Mais que peut-on faire pour la victime ?

La victime (la transaction, et donc le programme d'application) n'est normalement pas en tort. Il semblerait un peu dur de terminer purement et simplement le programme (quoique certains systèmes le font [DATE83]).

Certains systèmes, IMS par exemple [DATE83], chargent automatiquement une nouvelle copie du programme à laquelle ils donnent les mêmes messages en entrée qu'avant la rupture du deadlock. L'application peut ainsi tenter de se réexécuter convenablement. Un tel dispositif est bien entendu pratique dans un environnement de travail *on-line* [2].

D'autres systèmes renvoient simplement un signal à la victime ("il y a eu un deadlock et nous avons dû vous défaire"), sans pour autant stopper l'application. Le programmeur de l'application prévoit généralement le traitement de ce signal et peut donc envisager de relancer la transaction défaite. Ce procédé, appliqué par exemple par System R, est pratique dans un environnement *batch* [3].

Le redémarrage d'une transaction ("restart" en anglais), quelle que soit la méthode de contrôle de concurrence impléméntée et quelle que soit la raison du redémarrage, celui-ci se fera généralement après un délai aléatoire. Ce type de délai aurait en effet la propriété de stabiliser les situations dans lesquelles la propension des conflits deviendrait importante.

3.5.2.4. La prévention

Après avoir expliqué les méthodes qui n'écartaient pas les interblocages mais les résolvaient lorsqu'ils étaient découverts (détection), nous présentons celles de prévention[4].

---

[1] Techniques responsables des COMMIT et ABORT des transactions (et donc aussi de l'exécution des READ ou WRITE) afin d'assurer la conservation de l'intégrité de la BD, même après une panne, notamment grâce au UNDO (défaire) et REDO (refaire). On comprend qu'une telle tâche soit un tel centre d'intérêt.

[2] Environnement dans lequel les transactions doivent s'exécuter en temps réel, sans que l'utilisateur ne soit gêné par une quelconque attente. Les transactions y sont généralement assez courtes, comme par exemple dans les systèmes de réservation aérienne.

[3] Environnement dans lequel les transactions peuvent s'exécuter en différé, généralement sans la présence de l'utilisateur. Ces applications peuvent soit exécuter de longues séquences de transactions soit exécuter des transactions très longues. C'est ce qui explique que la technique du redémarrage complet du programme ne soit généralement pas une solution acceptable.

[4] Dans la littérature, on assimile fréquemment "avoidance" à "prévention". Certains auteurs précisent toutefois la notion d'avoidance comme étant plus radicale que celle de prévention du fait qu'elle agit plus tôt que cette dernière. La méthode radicalement conservatrice d'imposer à une transaction d'acquérir tous ses verrous ou aucun en est un exemple.

Ces dernières sont d'un usage moins courant, surtout dans les systèmes centralisés, mais leurs principes méritent d'être expliqués, d'autant plus qu'ils serviront de référence pour la présentation de leurs équivalents pour les BD distribuées.

Les interblocages étant l'apanage des techniques avec verrouillage, nous ne parlerons pas ici des techniques sans verrouillage. De fait, nous consacrons à chacune d'elles un point spécifique ultérieurement dans notre travail.

Nous avions répertorié quatre conditions nécessaires d'interblocage. Puisqu'elles devaient se présenter simultanément, en garantissant qu'une de ces conditions ne peut se réaliser, nous pouvons prévenir les interblocages. Nous verrons ci-dessous quelques politiques permettant de contourner le problème.

En considérant la première condition d'interblocage, l'exclusion mutuelle, on peut déduire que des ressources "partageables" n'en provoqueront pas puisque tous les accès pourront être garantis. C'est le cas par exemple des BD de simple consultation. Imposer cette condition est cependant presque toujours impossible pour des applications classiques.

1° Rejet de la demande

La première méthode est dite du "**rejet de la demande**". Il s'agit de vérifier si, en acceptant une demande de verrouillage d'une transaction, une situation d'interblocage sera provoquée[1] (une optique radicale serait de ne jamais accepter qu'une transaction attende; les interblocages sont alors évités mais au prix d'un grand nombre de transactions défaites).

Le scheduler devra donc vérifier si la demande de verrouillage d'une transaction doit être mise en attente et si l'arc qu'elle amène dans le WFG provoque un cycle quelconque.
Si nous prenons T_i la transaction qui demande un verrou sur une donnée X et T_j celle qui détient déjà un verrou conflictuel sur ce même objet, alors se pose le choix de choisir la victime : T_i ou T_j ? Le choix de T_i donne lieu à un algorithme de type **sans-préemption**, celui de T_j à un algorithme de type **avec-préemption**.

Remarque :

La version conservatrice du 2PL exigeait de connaître l'ensemble des opérations d'une transaction avant d'en exécuter ne serait-ce qu'une. Le scheduler pouvait ainsi essayer d'obtenir tous les verrous (avant de traiter les opérations proprement dites); en cas d'échec, il lui suffisait de libérer tous les verrous posés par cette transaction et mettre cette dernière en file d'attente. Si par contre tous les verrous avaient pu être accordés,

[1] Rappelons à cet effet que si une demande peut-être satisfaite, alors elle ne saurait provoquer un interblocage immédiat. Celui-ci n'a lieu qu'à cause d'une demande qui est mise en attente et qui provoque un cycle dans le WFG, graphe des mises en attente.

alors la transaction pouvait s'exécuter jusqu'à la fin sans le moindre risque de provoquer un interblocage.

Cette variation du 2PL, qui est une manière d'assurer que la condition "détient et attend" nécessaire à l'interblocage ne se réalisera pas, réduisait fortement la concurrence potentielle puisque les verrous étaient acquis plus tôt qu'il ne l'auraient été dans la version générale du 2PL. En fait, il n'existe sans doute aucune stratégie efficiente pour transformer un ensemble de transactions 2PL en un ensemble de transactions 2PL exempte d'interblocages sans réduire de manière inutile la concurrence potentielle.

2° Planification des transactions

La deuxième méthode est connue sous la dénomination de "**planification des transactions**". Il s'agit de planifier les exécutions des transactions de façon à ce que deux transactions ne soient pas exécutées en parallèle si leurs opérations sont conflictuelles.

Cette mesure évite évidemment tout risque d'interblocage mais,

a) elle exige de connaître à l'avance les données accédées par chaque transaction (c'est-à-dire qu'il doit y avoir prédéclaration[1] de ces données), et

b) comme il n'est pas toujours possible de déterminer exactement ces données, si ce n'est au moment de l'exécution, cette attitude sera souvent inutilement pessimiste et à concurrence affaiblie.

La technique de l'**ordonnancement préalable des ressources** est une variante de ce principe de planification des transactions. Elle exige aussi de connaître les listes d'intention des transactions (afin de pouvoir obtenir tous les verrous avant l'exécution d'une opération quelconque).

Les données sont numérotées et chaque transaction demande les verrous un à la fois dans l'ordre croissant des numéros. La priorité d'une transaction est donc donnée par le numéro de verrou le plus élevé qu'elle possède. Le fait de devoir obtenir les verrous séquentiellement est, en plus d'exiger la prédéclaration, un grief important contre cette méthode. Elle a en effet tendance à allonger sensiblement les temps de réponse.

3° Méthodes prioritaires : l'estampillage

La troisième méthode est liée aux méthodes basées sur les priorités. Dans un système de type prioritaire, le TM attribue à chaque transaction une priorité et décide en fonction de celle-ci si une transaction peut en attendre une autre ou si elle doit être avortée.

---

[1] Voir la sous-section 3.3.1 consacrée à la prédéclaration.

Plus précisément, on accepte de mettre une transaction T_i en attente d'une autre T_j si T_i est plus prioritaire que T_j auquel cas on ajoute aussi in arc de T_i vers T_j dans le graphe des mises en attente; sinon, T_i est avortée.

Un arc de T_i vers T_j signifie donc que T_i est plus prioritaire que T_j. Il sera par conséquent impossible d'arriver à un cycle parmi les transactions en attente l'une de l'autre puisqu'un cycle dans le WFG reliant une transaction à elle-même signifierait qu'elle est plus prioritaire qu'elle-même.

Le système prioritaire tel qu'il vient d'être défini prévient les interblocages mais pose néanmoins un problème nouveau : celui de voir une ou plusieurs transactions perpétuellement défaites. Cette situation, que nous examinerons plus en détail plus loin, est connue sous le nom de "famine".

L'exemple qui suit est destiné à illustrer ce problème et permettra d'introduire une variante du principe des priorités, l'estampillage, qui ne produit plus ce type de situations.

Supposons que nous ayons deux transactions, T_1 et T_2 qui souhaiteraient chacune exécuter les instructions suivantes :

T_1 : TR_BEGIN_1	T_2 : TR_BEGIN_2
$TR\_WRITE_1 (A, V_{11})$	$TR\_WRITE_2 (B, V_{21})$
$TR\_WRITE_1 (B, V_{12})$	$TR\_WRITE_2 (A, V_{22})$
TR_COMMIT_1	TR_COMMIT_2

Et imaginons qu'elles s'exécutent de la façon indiquée par la figure 3.4 ci-dessous :

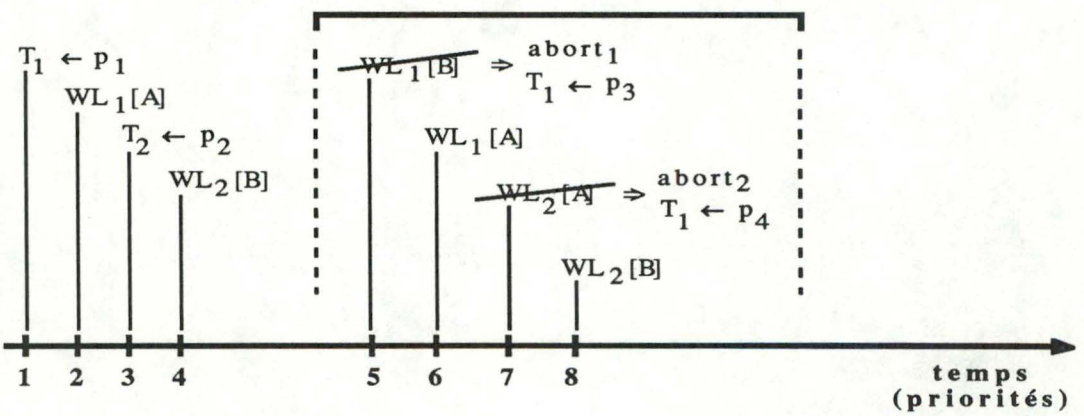


Figure 3.4 : Exemple de famine dans un système prioritaire

La transaction T_1 se voit attribuer par le TM une priorité p_1 et par le scheduler le verrou $WL_1[A]$; de même, T_2 reçoit ensuite une priorité p_2 et le verrou $WL_2[B]$. Ce sont respectivement les points 1, 2, 3 et 4 dans le temps.

Par après, le scheduler ne peut satisfaire la demande de $WL_1[B]$ qui entrerait en conflit avec le $WL_1[B]$. Par application du principe des priorités, $p(T_1)=p_1$ étant inférieure à $p(T_2)=p_2$, T_1 doit être défaite et reçoit une nouvelle priorité p_3 du TM de sorte qu'elle parvient à nouveau à obtenir le $WL_1[A]$.

Le même raisonnement peut être fait pour T_2 qui doit aussi être défaite car $p(T_2)=p_2$ est maintenant inférieure à $p(T_1)=p_3$. Elle reçoit une nouvelle priorité p_4 et le $WL_2[B]$ lui est accordé. Ce sont les étapes 5, 6, 7 et 8.

Ces quatre dernières étapes pourraient ainsi être recommencées sans fin, créant une situation de famine pour T_1 et T_2.

Cette situation est due au renouvellement des priorités après que les transactions aient été défaites. La variante présentée ci-dessous utilise des priorités, les estampilles qui ne sont pas renouvelées et ne provoquent donc pas ce type de problème.

L'estampillage est une méthode importante dans le contrôle de concurrence qui peut être utilisée à part entière comme une alternative au verrouillage (voir infra, le chapitre dédié à ce point) mais qui peut aussi servir de base à une méthode de prévention des interblocages que nous présentons maintenant.

Nous faisons de suite remarquer que la prévention par estampillage, à la différence de l'estampillage proprement dit est une technique avec verrouillage !

L'objectif de la prévention par estampillage n'est pas de prévenir les interblocages par des examens du WFG, mais de les éviter par le principe même du protocole imposé. Cette méthode cherche par là à minimiser la surcharge de travail tout en atteignant l'objectif de prévention (nous avons vu que cette surcharge était considérable). Elle utilise à cet effet un type de priorités particulières appelées *estampilles*.

L'estampille est un numéro d'ordre que le TM attribue à chaque transaction. Cette estampille[1] identifie toutes les transactions actives à ce moment-là. De plus, elle reste attachée à la transaction même si celle-ci est défaite et redémarrée en cours d'exécution.

Lorsqu'une transaction souhaite obtenir un verrou sur un objet qui est déjà bloqué par une autre transaction, on compare les estampilles de chacune et on prend une décision en

[1] Il s'agira généralement de l'"heure de démarrage" de la transaction ou du moins d'une variante garantissant l'unicité de l'estampille.

conséquence. Il existe deux manières de prendre cette décision qui donnent lieu à deux versions de l'estampillage : *Wait-Die* et *Wound-Wait* [1].

Supposons T_1 la transaction qui emet une demande de verrouillage et T_2 celle qui détient déjà un verrou sur le même objet.

Version **Wait-Die** :

si T_1 est plus âgée que T_2 :
- alors T_1 attend (T_1 - Wait)
- sinon T_1 est défaite et relancée (T_1 - Die)

Dans cette version, les transactions jeunes sont favorisées puisque les plus âgées ont tendance à attendre de plus en plus de jeunes au fur et à mesure qu'elles vieillissent.

La transaction possédant le verrou n'étant jamais défaite, il s'agit d'un algorithme "sans-préemption".

Version **Wound-Wait** :

si T_1 est plus âgée que T_2 :
- alors T_2 est défaite et relancée (T_1 - Wound)
- sinon T_1 attend (T_1 - Wait)

Dans cette version, les transactions âgées dépassent les plus jeunes (en essayant[2] de les tuer). La transaction détenant le verrou pouvant être défaite, il s'agit ici d'un algorithme "avec-préemption".

La manière Wait-Die a l'avantage sur la Wound-Wait qu'une fois tous les verrous acquis pour une transaction, elle est certaine de ne plus être défaite pour résoudre un interblocage (puisqu'elle ne peut être avortée qu'à cause de sa propre demande, alors qu'en Wound-Wait, elle l'est à cause d'une demande extérieure).

Wait-Die présente cependant l'inconvénient par rapport à Wound-Wait qu'une transaction défaite risque d'être réexécutée et à nouveau d'être avortée à cause de la même transaction.

---

[1] Chaque verbe exprime ici le point de vue de la transaction T_i qui formule la demande conflictuelle. En anglais, le "Wait" signifie que T_i attend, le "Die" que T_i est défaite et le "Wound" que T_i blesse T_j.

[2] En "essayant" car il se peut que la transaction à tuer ait déjà émis un TR_END (COMMIT ou ABORT), mais peu importe car dans ce cas les verrous sont tout de même relâchés et l'interblocage donc impossible.

Outre le fait de résoudre les conflits avant qu'il n'aient lieu, il est possible de prouver que ces techniques de prévention à base d'estampillage garantissent que chaque transaction se terminera (à moins d'une faute inhérente au programme) en des temps finis. En effet, étant donné la stricte croissance des estampilles et leur non-renouvellement, si une transaction reste suffisamment longtemps dans le système, elle finira par devenir la plus vieille. Dans aucune des deux méthodes une vieille transaction n'étant jamais tuée, celle-ci finira par être exécutée.

Il existe bien sûr d' autres techniques de prévention de deadlocks. Parmi celles-ci, les plus courantes procèdent à une analyse complexe de graphes avec insertion d'opérations de verrouillage-déverrouillage à l'insu des transactions. Lausen et al. [LAUS85] proposent une politique de verrouillage de ce type, prévenant les interblocages, le PAL (Pre-Analysis Locking). Ils évaluent leur méthode et les versions préventives du 2PL, et en concluent que chacune a ses avantages et inconvénients[1] mais qu'étant donné la différence fondamentale du niveau conceptuel auquel chacune travaille[2], une comparaison détaillée n'est pas possible. C'est ce qui rend d'ailleurs les deux politiques presque totalement incompatibles.

3.5.3. Les livelocks

Le livelock ou **famine**[3] est une situation dans laquelle une transaction ne parvient plus à obtenir certaines ressources pendant une durée anormalement longue. Nous verrons quelles sont les causes de ce bloquage et comment on peut s'en sortir.

Une transaction T_i peut être bloquée de façon excessive pour plusieurs raisons :

- ◊ T_i a une priorité d'exécution trop faible par rapport à d'autres transactions qui sont fréquemment utilisées;
- ◊ T_i ne peut s'exécuter de par certaines priorités de blocage;
- ◊ T_i attend une autre transaction T_j qui ne se termine pas (boucle infinie par exemple).

Wiederhold [WIED83] cite la circularité dans les contraintes d'intégrité comme une autre cause possible de famine.

Dans le premier cas, le système d'exploitation attribue des priorités aux différents processus qu'il exécute selon des critères variables (importance de l'utilisateur du système, importance de la tâche qu'il effectue, volume et type d'accès à certaines ressources - disque

---

[1] De façon assez succincte, alors que le 2PL verrouille plus tôt que nécessaire, le PAL essaie de verrouiller le plus tard possible et de déverrouiller le plus tôt possible.

[2] En 2PL, les opérations de (dé)verrouillage sont des actions sur des entités alors qu'en PAL, ce ne sont que des éléments de syntaxe.

[3] "Starvation" en anglais.

et CPU par exemple - , etc...). On comprend aisément qu'un nombre trop important de ce type de processus peut provoquer la famine d'autres processus à priorité trop faible.

Si la technique prioritaire ne peut être évitée, une manière de se sortir de cette situation serait de réviser (augmenter) périodiquement les priorités des processus en cours. De cette façon, un processus restant bloqué trop longtemps finirait par obtenir une priorité suffisante que pour pouvoir enfin se terminer.

Ces techniques ne sont pas du ressort du gérant des transactions du SGBD, mais plutôt du gérant des priorités du système d'exploitation. Les systèmes transactionnels auront généralement tendance à éviter ces types de schéma prioritaire.

Dans le second cas, les priorités ne sont pas générées par le système d'exploitation mais par le TM du SGBD. Nous avons déjà vu dans le cadre de la prévention des interblocages par priorités une illustration de la famine due au renouvellement des priorités et la solution à ce problème qui était l'estampillage. Ces priorités étaient explicites, mais il est des cas de figure où certaines transactions sont implicitement privilégiées par rapport à d'autres.

L'exemple suivant permet de comprendre rapidement d'où vient le problème :

Supposons que nous ayons une file d'attente des demandes de verrous (sur la figure 3.2, on peut assimiler la file d'attente de verrou à celle gérée par le scheduler pour la mise en attente des autres opérations, bien que ce soit normalement le gérant des verrous qui s'en charge). Dans cette file, quatre transactions attendent qu'une cinquième libère un verrou conflictuel.

Soit :

T_1 la transaction bloquante (elle possède un verrou $WL_1[X]$);

T_2 -> T_5, les transactions bloquées (elles ont respectivement émis les demandes $RL_2[X]$, $RL_3[X]$, $WL_4[X]$ et $RL_5[X]$).

Dès que T_1 se terminera, les demandes de T_2 et T_3 pourront être satisfaites. Pas celle de T_4 car elle entrerait en conflit avec celle de T_2 et T_3. Par contre, si le sceduler a pour politique d'essayer de satisfaire un maximum de transactions, alors il accordera aussi le verrou $RL_5[X]$ qui lui n'est pas conflictuel avec $RL_2[X]$ et $RL_3[X]$. Cette politique pourra donner lieu à la postposition perpétuelle de $RL_4[X]$.

Nous voyons par cet exemple comment, en voulant satisfaire un maximum de transactions de façon inconsidérée, on en arrive à provoquer la famine d'une ou plusieurs autres.

Ce danger peut être écarté de manière classique en suivant une politique FIFO (First In First Out, premier entré premier sorti), évitant ainsi qu'une quelconque transaction ne saute au-dessus d'autres.

Si on veut toutefois conserver l'avantage du maximum de transactions servies, on pourra, moyennant une légère surcharge, autoriser une transaction à sauter au-dessus d'une autre seulement si cette dernière n'attend pas depuis "trop longtemps" (ce délai devra bien sûr être choisi judicieusement).

Ce cas permet aussi de comprendre pourquoi une transaction en vient à être défavorisée par rapport à d'autres. Dans l'exemple ci-dessus, T_4 était, relativement, trop exigeante puisqu'elle requérait un verrou exclusif alors que les autres ne demandaient que des verrous partagés, et c'est pourquoi elle a subi ce sort.

On peut en déduire que d'autres cas de ce genre se produiront. Si on prend par exemple le cas de la prédéclaration (voir le point consacré aux schedulers de type 2PL conservateur), une transaction ayant un WriteSet trop volumineux, c'est-à-dire l'ensemble des données qu'elle veut modifier étant trop grand, elle risque aussi de se voir constamment retardée, le scheduler lui préférant des transactions à prédéclarations plus restreintes. Les transactions longues (ou du moins accédant, surtout en écriture, à de nombreux enregistrements) sont donc celles qui patissent le plus de ce problème. A nouveau, il faudrait modifier le scheduler de façon à ce que pareille situation soit évitée.

Le dernier cas, celui d'une transaction attendant la libération d'un verrou détenu par une transaction bouclante, trouve normalement déjà une solution grâce au mécanisme de détection d'interblocages du scheduler. La transaction en famine se verra débloquée lorsque la transaction bloquante sera soit défaite soit satisfaite suite à la mise en oeuvre du mécanisme sus-mentionné.

Notons que Bayer et al. [BAYE76] font une proposition qui ne manque pas d'intérêt destinée à éviter les situations de famine : essayer, au sein d'un même système, d'utiliser différentes stratégies successivement de façon à garantir qu'une famine non détectée par une méthode le soit par la suivante. L'effectivité en ressort augmentée mais évidemment aux dépens de stratégies parfois coûteuses.

3.6. La granularité

Une question importante lors de la conception d'un système de gestion de bases de données est le choix des unités verrouillables, c.-à-d. de l'agrégat de données qui sera verrouillé afin d'assurer la cohérence.

Ce verrouillage peut avoir lieu à deux niveaux : à un niveau logique et à un niveau physique.

Ce que l'application demande tient du niveau logique; ce que le SGBD implémente tient du niveau physique. L'application parle souvent en termes d'enregistrements alors que le scheduler du SGBD utilise plutôt la notion de page. Il pourra parfois y avoir correspondance entre les deux niveaux (verrouillages logique et physique tous deux au niveau de l'enregistrement par exemple). Mais dans d'autres cas, l'inédaquation sera flagrante (verrouillage logique des enregistrements, verrouillage physique de tout le fichier).

Notons de suite que cette considération n'a aucune influence sur la cohérence de la BD. Elle n'influence que les performances qu'on peut en espérer. Que penser en effet d'un SGBD qui offre au programmeur d'application des verrous sur les enregistrements mais qui réalise ce verrouillage sur toute la page contenant l'enregistrement ?

Nous présentons ci-dessous les notions de verrouillage physique et de verrouillage prédicat (qui est un verrouillage logique). Pour l'un et l'autre, nous développons les raisons d'être et les difficultés de réalisation.

3.6.1. Le verrouillage prédicat

On parle de **verrouillage prédicat** lorsqu'il est possible de poser un verrou logique sur une portion exacte de la BD. Cette portion à verrouiller est déterminée par un prédicat ou une qualification. On peut par exemple verrouiller "*tous les comptes de clients dont le montant au débit est inférieur à 50000 unités* ". Ce prédicat ne verrouille pour la transaction qui le formule qu'un sous-ensemble logique de la BD.

Le verrouillage prédicat est donc une extension du verrouillage individuel d'objets communément offert par les SGBD.

Il n'y aura que très rarement correspondance parfaite entre les objets logiques qu'on demande de verrouiller et les objets physiques qui seront réellement verrouillés.

Cette non-correspondance constitue une des difficultés majeures pour les systèmes offrant la technique du verrouillage prédicat : s'ils n'offrent que des prédicats élémentaires, de tels systèmes ne sont pas très attrayants; par contre, s'ils prétendent offrir le traitement de prédicats complexes, il y a lieu de vérifier si ces systèmes ne verrouillent bien que les objets qui satisfont au prédicat et pas, par exemple, toute la BD.

Mais la difficulté majeure du verrouillage prédicat est de détecter les conflits, car cela nécessite une vérification de la compatibilité de tous les verrous à poser pour ce prédicat avec tous ceux déjà posés par tous les autres prédicats. Le problème est en effet réputé récursivement insolvable (c'est-à-dire insolvable par un algorithme pour TOUS les cas de figure possibles[1]), même si on force les prédicats sont limités à l'utilisation des seuls opérateurs arithmétiques +, *, -, /. Les implémentations existantes doivent donc restreindre les prédicats à une forme suffisamment simple que pour permettre cette recherche de conflits. Les prédicats simples acceptés seront généralement des combinaisons de prédicats atomiques du type : <nom_de_champ> op_comp <constante>, où op_comp est un opérateur de comparaison.

Le verrouillage prédicat pose aussi une nouvelle question : celle des données "phantômes" (fig. 2 de l'introduction). Pour une transaction T donnée, il faut en effet verrouiller non seulement toutes les données satisfaisant au prédicat, mais aussi toutes celles qui n'existent pas encore, pourraient être créées par d'autres transactions et satisfaire au prédicat avant que T ne soit terminée. Il faut donc verrouiller ce que Eswaran et al. [ESWA76] appellent la "non-existence" des données du prédicat à verrouiller; ils proposent un protocole de verrouillage prédicat basé sur le 2PL comme solution à ce problème des données phantômes.

3.6.2. Le verrouillage physique

Dans le cas du **verrouillage physique**, on verrouille au niveau des fichiers, des segments, des pages, des enregistrements, etc. La littérature désigne ces unités verrouillables sous le nom de "granules" de la BD. Il faut bien se rendre compte que le SGBD devra être prêt à gérer une table de verrous avec autant d'entrées qu'il y a de granules dans la BD.

Le choix des unités de verrouillage est donc un compromis entre la concurrence et la surcharge amenée par la gestion de ces verrous, laquelle est précisément liée à cette taille ou "granularité" des verrous.

D'un côté, des unités réduites de verrouillage augmentent la concurrence, puisqu'elles diminuent la probabilité d'apparition d'un conflit. Par exemple, dans le cas de granules correspondants à des enregistrements de la BD, les transactions pourront s'exécuter en parallèle (sans conflit) aussi longtemps qu'elles accèderont à des enregistrements distincts. De telles unités sont bien adaptées au cas des transactions "simples", accédant à peu d'enregistrements. D'un autre côté, elles seraient coûteuses dans le cas d'une transaction "complexe", accédant à un grand nombre d'enregistrements, puisque la transaction devrait

[1] A fortiori dans un environnement distribué.

sans cesse poser et reposer des verrous, surchargeant en travail et en espace le gérant des verrous.

Par contre, de larges unités de verrouillage sont certainement pratiques pour une transaction "complexe", mais elles sont discriminantes pour des transactions qui ne doivent accéder qu'à peu d'enregistrements. Si le granule considéré est la BD entière, le gérant n'aura à surveiller qu'un seul verrou, mais toute concurrence sera impossible (ou presque).

Ries et Stonebraker [RIES77] donnent quelques exemples concrets de granules : l'enregistrement pour CODASYL, System R ou DMS-1100; la page pour IMS; la colonne d'une relation pour INGRES; tous les enregistrements d'un type donné pour LSL; toute la BD pour SYSTEM 2000; etc. System R propose même un granule dont la taille peut varier dynamiquement. C'est pourquoi, dans cette optique, nous proposons à la section suivante (3.7), un protocole de verrouillage pour un système où coexistent différentes granularités.

Ries et Stonebraker y analysent aussi les impacts de la granularité dans un SGBD : construction d'un modèle fortement paramétrable, simulations et conclusions. Nous retiendrons notamment que dans le cas du verrouillage prédicat, c'est le nombre de transactions actives qui détermine le nombre de verrous, et non la taille de la BD.

StoneBraker [STON84] étudie également les conséquences d'un système d'exploitation qui offrirait la possibilité de travailler en transactionnel en mémoire virtuelle. Une telle possibilité impose la page comme "granule" le plus fin, c'est-à-dire que deux transactions ne peuvent mettre à jour la même page sans mettre en cause la cohérence du système.

Ce ne sont donc pas les débats qui manquent en matière de granularité acceptable ou non, ni les preuves. Le choix de la granularité du verrouillage est bel et bien un compromis.

3.7. La multi-granularité (ou hiérarchie des verrous)

Jusqu'à présent, nous n'avions fait aucune supposition sur la structure de la BD. Elle n'était qu'un ensemble désordonné de données. En réalité, ces données se trouvent stockées sur des supports physiques, qui, en tant que telles, peuvent être hiérarchisés[1] comme la figure 3.5 nous en donne l'exemple [GRAY79], [BERN87]. Il ne s'agit pas là d'un schéma purement logique de la BD mais d'un schéma hybride repésentant la répartition physique des ressources logiques. Ainsi, la BD est répartie sur plusieurs *areas* (régions de disques), contenant elles-même plusieurs fichiers regroupant des enregistrements.

[1] C'est-à-dire un arbre non dégénéré : un noeud a au plus un parent.

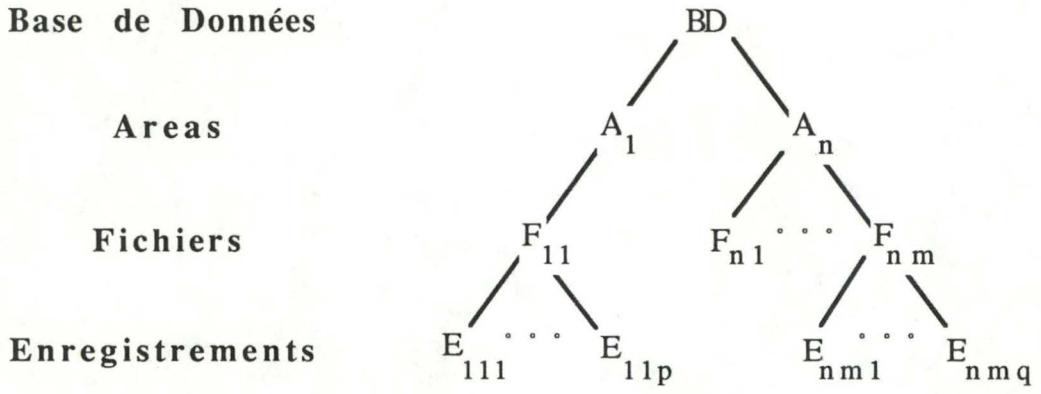


Figure 3.5 : Exemple de hiérarchie de verrouillage

Comme nous l'avions déjà dit plus haut, il peut être pratique de disposer d'un SGBD offrant une possibilité de verrouillage avec différentes granularités (la littérature parle parfois de **MGL**, Multi-Granularity Locking). Dans la hiérarchie ci-dessus (fig. 3.5), chaque noeud pourrait donc être verrouillé, de la granularité la plus fine - l'enregistrement - à la plus grossière - la BD.

La cohérence n'est nullement concernée par cette implémentation; on ne recherche par là qu'à améliorer les performances d'accès à la BD, ce qui ne fait que rendre la tâche du SGBD plus complexe. Nous verrons par la suite quels sont les problèmes amenés par cette facilité et comment ils peuvent être résolus.

Rappelons que nous souhaitions [1] *établir un équilibre entre la surcharge amenée par un nombre trop important de "petits" verrous et la perte de concurrence provoquée par des verrous "grands" mais peu nombreux. L'idée était donc de se donner les moyens de choisir la "taille" (granularité) des verrous en fonction des besoins.*

De longues transactions, accédant à de multiples enregistrements d'un fichier, pourraient ainsi réduire le nombre de verrous à poser en verrouillant par exemple tout le fichier. De même, les petites transactions, ne touchant qu'à quelques enregistrements locaux, pourraient les verrouiller individuellement, évitant ainsi de bloquer de grands morceaux de la BD auxquels ils n'accèdent de toutes façons pas.

Avec des ressources structurées d'une manière hiérarchique, si un demandeur obtient le verrouillage en mode X (pour eXclusif) d'un noeud particulier, alors il gagne un accès exclusif non seulement à ce noeud mais implicitement à tous les descendants de ce noeud

[1] Voir la section 3.6 consacrée à la granularité et à ses conséquences.

également. Par "implicitement", nous entendons que ce n'est pas du ressort de la transaction, mais bien du scheduler. Par "explicitement", nous signifions donc une véritable demande de verrouillage de la part d'une transaction.

De même, si un demandeur obtient le verrouillage d'un noeud en mode S (pour Shared), alors il l'a pour ce noeud ainsi que pour tous ses descendants. C'est là l'essentiel du problème de la multi-granularité : un verrou peut porter sur tout un sous-arbre du graphe.

Les techniques du type MGL devront donc éviter que des transactions différentes ne verrouillent de manière incompatible des granules qui se chevauchent (en tenant compte du caractère implicite de cette action).

Les deux modes d'accès (X et S) que nous avions admis jusqu'à présent ne suffiront plus . En effet, afin de verrouiller un sous-arbre dont la racine est R, il est crucial d'éviter des verrous sur les ancêtres de R et qui pourraient verrouiller R et son sous-arbre de façon conflictuelle[1].

C'est pourquoi nous devons introduire un nouveau mode d'accès aux données, le *mode d'intention I*.

3.7.1. Le mode d'intention

Le **mode d'intention** a pour objectif de repérer et marquer tous les ancêtres d'un noeud à verrouiller. Grâce à ces marques, le scheduler[2] est averti que des verrous ont été posés à un niveau inférieur. Ces repères lui permettent d'empêcher un conflit (explicite ou implicite) lors du verrouillage des ancêtres. Il y a donc propagation des effets des verrous les plus fins vers les verrous les plus grossiers qui les contiennent.

C'est pourquoi on a étendu les deux modes existants au moyen de trois nouveaux : **IS** (Intention Shared), **IX** (Intention eXclusif) et **SIX** (Share and Intention eXclusif). Il s'agit respectivement des modes d'intention partagée, exclusive et enfin partagé et d'intention exclusive.

Conformément à une notation déjà définie antérieurement, IRL[X] signifiera un verrou en mode Intention Read sur la donnée X (c'est-à-dire IS[3]), et IWL[X] un verrou en mode Intention Write sur X (c'est-à-dire IX[4]).

---

[1] Par exemple : un enregistrement verrouillé en écriture par une (petite) transaction devrait empêcher une autre (plus grande) d'obtenir un verrou, même de lecture, sur le fichier qui contient cet enregistrement. Une solution serait que la plus longue transaction examine tous les enregistrements du fichier en question, à la recherche d'éventuels verrous. On constate aisément le côté absurde de cette issue puisqu'elle ne ferait qu'alourdir considérablement la gestion des verrous plus "fins" alors que c'est ce qu'on cherche à éviter.

[2] Le graphe de la figure 3.4, rappelons-le, n'est qu'un modèle utilisé par le scheduler afin d'assurer la gestion des verrous de différentes granularités.

[3] La littérature parle indifféremment de S (Shared) et de R (Read) puisqu'il s'agit de la même chose : un accès en lecture est un accès qui peut être partagé par plusieurs transactions.

[4] Même remarque que ci-dessus : un accès en écriture W (Write) est un accès qui doit rester exclusif (X) à une transaction.

Les modes IS et IX ont bien sûr été créés sur base de la raison d'être du mode d'intention (que nous venons d'expliquer). Nous pouvons expliciter cela à l'aide de l'exemple suivant :

Supposons que le scheduler veuille satisfaire une demande de verrou RL[X]. Il doit garantir qu'aucun verrou sur un ancêtre de X ne provoquera de conflit implicite avec le futur RL[X]. Il devra donc marquer les ancêtres de X de verrous IR. Concrètement, si X est un enregistrement, le scheduler devra marquer d'un IRL les ancêtres de X, soit dans l'ordre la BD, l'area et le fichier contenant X. Puisque IRL et WL sont conflictuels, il empêchera ainsi tout WL sur un ancêtre de X qui verrouillerait implicitement X en écriture.

De même, si une transaction veut obtenir un WL[Y], les IWL étant incompatibles avec les RL ou les WL, le scheduler évitera tout conflit implicite en marquant les ancêtres de Y d'un IWL.

Le mode SIX trouve, lui, une justification un peu plus élaborée.

Supposons qu'une transaction souhaite parcourir tout un ensemble d'enregistrements afin de modifier certain d'entre eux présentant telle particularité. Cette tâche doit donc poser un RL sur tout un fichier (afin de pouvoir en lire tous les enregistrements) et un IWL (afin de pouvoir modifier les quelques enregistrements souhaités). La situation étant assez fréquente, un mode hybride SIX (aussi noté RIW) a été conçu[1]. Ce mode est un équivalent logique du mode S et du mode IX, puisqu'il permet un accès partagé au sous-arbre complet sans autre verrouillage et aussi un accès d'intention exclusive. Ce dernier donne à la transaction la possibilité de verrouiller en mode X les noeuds à modifier, et en mode IX ou SIX les autres noeuds.

Nous avons donc les **modes** suivants :

S (et **X**) : donnent un accès partagé (exclusif) au noeud demandé et implicitement à tous ses descendants; ces derniers ne doivent donc plus être verrouillés;

IS (et **IX**) : donnent un accès d'intention partagée (exclusive) au noeud demandé et autorisent les verrous S ou IS (S, X, IS, IX ou SIX) sur les descendants; ces derniers ne sont donc pas verrouillés implicitement;

[1] Sans l'existence de ce mode, deux solutions basées sur les autres modes permettaient de résoudre ce problème : **a)** brutalement verrouiller la racine du sous-arbre en mode X ou **b)** demander un accès IX à la racine du sous-arbre et verrouiller explicitement les noeuds inférieurs désirés en mode I, S ou X. La solution a) réduit considérablement la concurrence, alors que la solution b) alourdit fortement la gestion des verrous, surtout si peu d'enregistrements sont mis à jour.

SIX : donne un accès partagé et d'intention exclusive au noeud demandé; les verrous sur les descendants sont implicitement partagés, mais peuvent être explicités en mode X, IX ou SIX.

Le tableau 3.3 ci-dessous offre une vue d'ensemble de la compatibilité des cinq types de verrou satisfaisant les besoins du MGL :

Tableau 3.3 : Compatibilité des verrous en MGL

Compatibilité ?		Donnée verrouillée en mode				
		S	X	IS	IX	SIX
Donnée demandée en mode	S	OUI	NON	OUI	NON	NON
	X	NON	NON	NON	NON	NON
	IS	OUI	NON	OUI	OUI	OUI
	IX	NON	NON	OUI	OUI	NON
	SIX	NON	NON	OUI	NON	NON

Nous pouvons maintenant clairement exprimer le protocole MGL que devront respecter les transactions. Nous avons en effet vu que certains types de verrous donnaient lieu à des verrouillages implicites au niveau des sous-arbres, ce qui fait que les transactions ne peuvent pas verrouiller n'importe quel noeud de façon incontrôlée. Nous avions par exemple déjà fait remarquer l'importance de l'ordre de pose des verrous (de haut en bas dans l'arbre). Le protocole MGL qui sera présenté ci-dessous permettra en outre de déduire que leur libération devra se faire dans l'ordre inverse.

Mais examinons encore une fois le tableau 3.3 donné plus haut. Il nous permet en effet de classer les cinq types de verrous selon un ordre partiel[1], par ordre décroissant de privilège de chaque mode :

$$X - \begin{matrix} S \\ SIX \\ IX \end{matrix} - \begin{matrix} et \\ \end{matrix} - IS$$

Un verrou X sur un noeud autorise en effet la modification de n'importe quel noeud de son sous-arbre sans que soit nécessaire un autre verrouillage.

[1] S et IX sont incomparables et ne permettent donc pas d'obtenir un ordre partiel.

Le verrou IS n'a lui que peu de poids dans la balance puisqu'il n'exclut que les verrous X et ne peut donner lieu qu'à un accès en lecture.

Cet ordre est important car il explique bien la "force" des verrous et certaines actions interdites par le **protocole MGL** que voici :

◊ avant de pouvoir verrouiller un noeud en mode S ou IS, une transaction doit déjà posséder un verrou de type IS ou supérieur sur le parent de ce noeud (sauf s'il s'agit de la racine globale de l'arbre, bien sûr);

◊ avant de pouvoir poser un verrou de type X, IX ou SIX sur un noeud, une transaction doit déjà posséder un verrou de type IX ou supérieur sur ce noeud (sauf à nouveau s'il s'agit de la racine globale de l'arbre);

◊ une transaction ne peut libérer un verrou d'intention sur une donnée si elle possède encore des verrous sur un quelconque des descendants de ce noeud[1].

Bernstein, Hadzilacos et Goodman prouvent l'exactitude de ce protocole dans [BERN87]. Ils soulignent toutefois que le MGL, bien que garantissant le non-conflit entre transactions, ne garantit nullement la sérialisabilité à lui seul. Il doit donc être utilisé en conjonction avec le 2PL. Ils esquissent la collaboration entre les deux techniques en disant que le 2PL prévoit *quand* poser les verrous alors que le MGL prévoit *comment* les poser.

Nous avons vu jusqu'à présent comment le scheduler devait théoriquement assurer le verrouillage afin d'éviter les conflits et de garantir la sérialisabilité des exécutions. Toutefois, afin de simplifier la tâche du programmeur d'application, le SGBD qu'il utilise devrait, pratiquement, décider lui-même de la granularité des verrous.

Pour ce qui est des verrous du niveau le plus fin, cette gestion est aisée puisqu'il suffit au scheduler de requérir ceux-ci un par un, au fur et à mesure des opérations en provenance des transactions.

Par contre, pour ce qui est des verrous à granularité plus grossière, le problème de la prévision resurgit.

Quelques solutions[2] ont déjà été envisagées, parmi lesquelles, l'analyse du profil récent des applications afin de prévoir au mieux le type d'opérations à venir. Cette méthode s'applique bien au cas de la multi-granularité. Elle pourrait s'énoncer comme suit : *commencer par verrouiller les données au niveau le plus fin; ensuite, si une proportion*

---

[1] Une transaction doit donc libérer ses verrous dans l'ordre inverse de la pose, c'est-à-dire des feuilles vers la racine, excepté si elle décide de les relâcher tous en une seule fois à la fin avant de se terminer.

[2] Nous avions en effet déjà signalé la difficulté de prévision du comportement futur d'une transaction. La prédéclaration était une solution - contraignante - à ce problème. Existait aussi une solution de pré-analyse de l'application au moment de la compilation et enfin une solution de prédiction dynamique consistant à étudier le profil des transactions lors de leur exécution.

"importante" de verrous fins ont été posés dans le cadre de la transaction, passer à un niveau de verrouillage plus grossier etc... jusqu'au niveau maximum si nécessaire. Cette méthode est connue sous le nom d'**escalade du verrouillage**.

Cette solution présente néanmoins le même inconvénient que la conversion de verrou : le risque d'interblocage. Etant donné la similarité des problèmes, nous les développons ensemble dans le point suivant.

3.7.2. L'escalade et la conversion de verrou en MGL

Nous avions vu, dans le cas de la granularité fixe, qu'une conversion de verrou du type R (lecture) -> W (écriture) permettait de parcourir un ensemble de données en mode partagé, afin de modifier telle ou telle donnée si elle satisfaisait à un critère déterminé.

Cette possibilité de convertir un verrou en cours de transaction garde tout son intérêt dans le cas du protocole MGL. Toutefois, étant donné la diversité des modes existants (S, X, IS, IX et SIX), cette gestion devient plus complexe.

Le scheduler peut déterminer les conversions légales en s'aidant soit du tableau 3.3 des compatibilités soit du schéma d'ordre partiel (cfr. supra).

Le scheduler pourrait, par exemple, appliquer la règle suivante (sur base du schéma d'ordre partiel) :

> *"à partir du type de verrou posé et du type de verrou demandé, la conversion à effectuer est donnée par le verrou le plus fort des deux, sauf s'il y a ambiguïté 1, auquel cas la conversion se fait au verrou du niveau supérieur ."*

Il reste encore à vérifier que la demande est compatible avec des verrous déjà détenus par d'autres transactions (si par hasard la conversion devait provoquer un conflit, elle serait mise en attente). Pour de plus amples explications, [GRAY79] consitue une bonne référence.

> *Les auteurs de [BERN87] proposent, afin d'accélérer au maximum la tâche du scheduler, d'implémenter non pas un algorithme semblable à celui ci-dessus mais bien une table fixe à deux entrées. On obtient alors la conversion à réaliser à l'intersection de la colonne du verrou posé et de la ligne du verrou demandé.*

Il existe dans le cadre de la MGL un problème qui lui est spécifique, celui de l'**escalade de verrou**. Il présente un risque tout à fait similaire à celui de la conversion de verrou, mais alors que la conversion avait lieu sur la demande des transactions, l'escalade

---

[1] Il s'agit des cas S et IX. Il faut alors convertir en SIX.

de verrou est une décision dynamique du scheduler. L'exemple suivant permet d'illustrer ce problème :

Prenons deux transactions ayant verrouillé le même fichier en mode IW (verrous compatibles). Toutes deux accèdent aux enregistrements en lecture, puis, pour certains enregistrements, décident d'effectuer une écriture. Supposons que le scheduler ait commencé à verrouiller ces enregistrements en mode W. Si, face au nombre croissant des mises à jour, il décide de passer à une granularité supérieure (escalade), il provoquera alors le deadlock. Chacune des deux transactions devra en effet attendre la libération du verrou IW posé par l'autre pour pouvoir poser le sien, W.

3.7.3. Graphe acyclique orienté : DAG

Nous avions supposé, lors de l'introduction à ce point consacré à la multi-granularité, que l'on pouvait structurer les ressources de manière hiérarchique selon un critère d'emplacement physique (fig. 3.5). Bien que raisonnable, cette hypothèse s'avère par trop restrictive dans certains cas.

L'exemple le plus évident est celui où un fichier d'accès séquentiel est muni d'une table d'index afin d'offrir un accès sur base d'une clé à ses enregistrements. Nous n'avons donc plus la hiérarchie de la figure 3.5, mais un arbre dégénéré (fig. 3.6) où chaque paire (fichier, index) descend d'une area[1] et où les enregistrements sont les descendants et d'un fichier et d'une table d'index[2] :

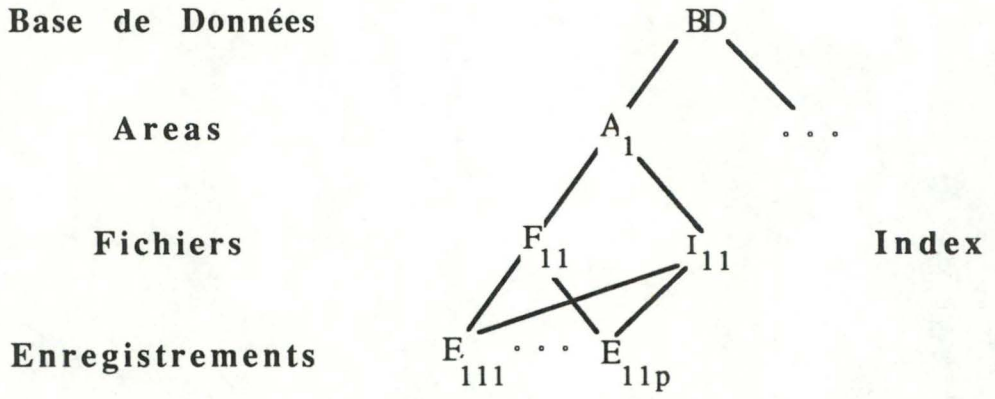


Figure 3.6 : Exemple d'arbre dégénéré

[1] A chaque fichier est associé une table d'index qui se trouve dans la même area.

[2] Un enregistrement est accédé soit séquentiellement grâce au fichier, soit directement grâce à l'index défini sur un champ particulier de cet enregistrement. Il est à noter que ce graphe n'est pas purement physique. L'enregistrement est bel et bien un descendant physique du fichier auquel il appartient, mais il est aussi lié logiquement à ce fichier par la table d'index qui lui est associée. C'est la fusion logique des deux qui conditionne l'existence des enregistrements.

Cet arbre dégénéré (fig. 3.6) porte généralement le nom de **DAG** (Directed Acyclic Graph), graphe orienté acyclique. Cette dégénérescence vient du fait de la double possibilité d'accès à chaque enregistrement : séquentiel (fichier) et direct (index).

C'est cette même propriété qui fera que ces DAG ne pourront être gérés de la même manière que les hiérarchies décrites ci-dessus. On désirerait par exemple pouvoir verrouiller des enregistrements à partir de leur entrée dans la table d'index. Comme il s'agit d'un graphe, il faudra envisager le verrouillage de tous les parents d'un noeud[1] selon le mode approprié.

Par exemple, dans le cas de la figure 3.6, pour modifier l'enregistrement E111 du fichier F11 et dans l'index I11, on pourrait verrouiller dans l'ordre : BD, A1, F11 et I11 en mode IX et E111 en mode X.

Nous proposons maintenant le protocole MGL pour un DAG. Il est analogue au premier à la différence essentielle près qu'un noeud peut avoir plusieurs parents.

Une transaction obtiendra un verrou S implicite sur un noeud si au moins un des parents de celui-ci a déjà été verrouillé, explicitement ou implicitement, en mode S, X ou SIX, et par conséquent au moins un des ancêtres verrouillé explicitement dans un de ces modes.
Elle pourra aussi obtenir un verrou X implicite sur un noeud si tous ses parents ont déjà été, explicitement ou implicitement, verrouillés en mode X.

Le **protocole DAG** que le scheduler devra respecter sera donc le suivant :

◊ pour pouvoir garantir un verrou S ou IS sur un noeud, il faut que la transaction demanderesse ait déjà un verrou de type IS ou supérieur sur au moins un des parents de ce noeud[2];

◊ avant de pouvoir accorder à une transaction un verrou de type X, IX ou SIX sur un noeud, il faut que celle-ci ait un verrou de type IX ou supérieur sur tous les parents de ce noeud;

◊ les verrous doivent à nouveau être relâchés dans l'ordre inverse d'acquisition à moins de ne l'être tous à la fois à la fin de la transaction.

Pour ce protocole assez délicat, tout comme pour le cas hiérarchique, la littérature ([GRAY79], [BERN87]) offre des exemples et des preuves de validité. Nous voulions simplement mettre en évidence les similitudes et différences avec le MGL; c'est pourquoi nous ne nous y attarderons plus.

---

[1] La structure hiérarchique est simplement un DAG où chaque noeud n'a qu'un seul parent.
[2] Et par conséquent, le long d'au moins un chemin menant vers une racine.

Signalons simplement que des protocoles qui ne sont pas de type 2PL (notés "non-2PL") définis sur des BD dans lesquelles les données peuvent être structurées selon un DAG ont été étudiés par Mohan et al. dans [MOHA85]. On y trouve également des généralisations de ces techniques, leurs preuves et certaines méthodes de résolution d'interblocages.

Chapitre 4

L'estampillage : TO

Les méthodes basées sur l'estampillage que nous présentons ici sont les premières méthodes que nous développerons et qui ne font pas appel au verrouillage. Nous présenterons aux chapitres suivants des méthodes basées sur des tests d'un graphe particulier, le graphe de sérialisation, et des méthodes dites optimistes.

4.1. La règle de l'estampillage

Nous avions déjà légèrement abordé le principe de l'estampillage lors de l'exposé relatif aux techniques de prévention d'interblocages à base d'estampillage, mais celles-ci utilisaient encore le verrouillage.

Le **principe de l'estampillage** proprement dit ("timestamping" ou TO pour "Timestamp Ordering" en anglais) est le suivant : à chaque transaction T_i, le scheduler attribue selon un ordre chronologique, une estampille $e(T_i)$ unique[1]. Si la transaction est défaite, alors elle se voit assigner une nouvelle estampille, contrairement aux estampilles utilisées pour la résolution des deadlocks qui, elles, n'étaient pas rajeunies.

La **règle de l'estampillage** est alors la suivante :

soit $op_i[X]$, une opération demandée par la transaction T_i
et $op_j[X]$, une opération demandée par la transaction T_j.

si les deux opérations sont en conflit,
alors

$op_i[X]$ devra être exécutée avant $op_j[X]$
ssi
$e(T_i) < e(T_j)$

[1] C'est le fait que l'estampille soit unique qui importe. Quoiqu'on lui associe souvent la notion d'"*heure de démarrage*" de la transaction, un simple compteur incrémenté à chaque nouvelle estampille est généralement suffisant et moins coûteux.

Le but est évidemment d'obtenir un classement des transactions selon un ordre strict (de manière notamment à éviter les interblocages), du moins lorsque les opérations sont conflictuelles. La preuve de la sérialisabilité des exécutions produites par le protocole du TO peut être trouvée dans [BERN87]. La différence essentielle en la matière est que les techniques de verrouillage synchronisaient les opérations des transactions de manière à arriver à une exécution équivalente à une quelconque exécution sériale, alors que les techniques d'estampillage le font de manière à arriver à une exécution sériale particulière, à savoir celle déterminée par l'ordre des estampilles.

Remarque :

L'estampillage est une méthode sans verrouillage. C'est la première que nous présentons avec cette particularité.

Maintenant que le principe de l'estampillage a été exposé, nous pouvons présenter les protocoles qui en ont résulté. Nous avons choisi les trois versions les plus représentatives à savoir, les versions de base, conservatrice et stricte.

4.2. Protocole de base

Avant de pouvoir soumettre au lecteur le premier algorithme basé sur l'estampillage, nous devons redéfinir la notion de conflit puisqu'il n'y a plus verrouillage.

Pour deux transactions T_1 et T_2, d'estampilles respectives $e(T_1)$ et $e(T_2)$ avec $e(T_1) < e(T_2)$[1], accédant au même objet O, il y aura **conflit**

si

T_1 lit O qui a déjà été modifié par T_2

ou si

T_1 modifie O qui a déjà été lu ou modifié par T_2.

C'est-à-dire que lorsque le scheduler reçoit une demande d'opération de T_i, il vérifie si cette opération n'entre pas en conflit avec une opération d'une transaction T_j qu'il aurait déjà envoyée au DM[2].

En cas de conflit, il ne lui reste que la solution de défaire et redémarrer la transaction qui a formulé la demande tardive (T_1 dans notre cas, même si elle est la plus jeune). Celle-ci se verra alors assigner une estampille plus grande (et s'en trouvera rajeunie) afin d'avoir plus de chances pour sa nouvelle exécution. Cet exemple est illustré plus clairement grâce à la figure 4.1 ci-dessous :

---

[1] $e(T_1) < e(T_2)$ signifie que T_2 est plus jeune que T_1.

[2] Le scheduler vérifie donc si cette demande n'arrive pas trop tard; cela suppose que l'ordre des opérations transmises au DM est respecté par celui-ci.

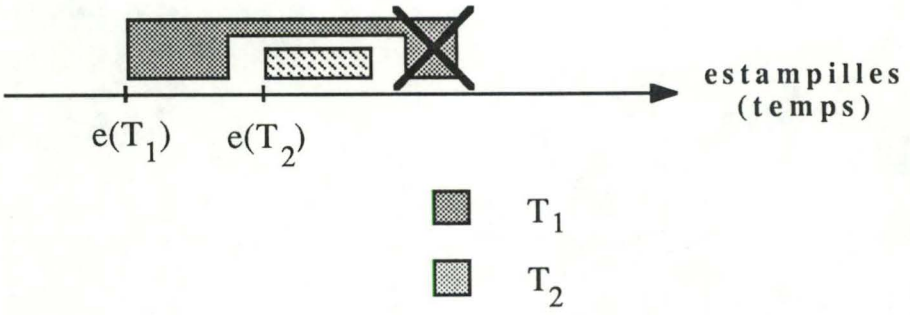


Figure 4.1 : Transaction trop âgée défaite

Pour les techniques de verrouillage, nous avons vu que les modifications d'une transaction pouvaient être amenées sur la BD réelle avant qu'elle ne se termine, ce qui provoquait lesdits problèmes de recouvrement en cas d'ABORT. Puisque les méthodes de TO ne posent pas de verrous, elles ne peuvent pas effectuer de mises à jour sur la BD réelle tant qu'elles ne sont pas arrivées à la fin de la transaction. Lors d'un COMMIT, elles seront toutes apportées en même temps (si possible[1]); lors d'un ABORT (et a fortiori, lors d'un restart), il ne sera jamais nécessaire d'effectuer le défaire à un niveau physique.

Toutefois, l'inconvénient majeur de cette technique est qu'elle exige un travail de journalisation volumineux. En effet, pour pouvoir déterminer si une transaction est arrivée trop tard, le TO (version de base) doit maintenir à jour, pour CHAQUE objet accédé O_k, deux données : l'estampille de la dernière transaction qui a réussi une lecture de O_k et l'estampille de la dernière transaction qui a réussi[2] une écriture, soit respectivement e_der_lec et e_der_maj.

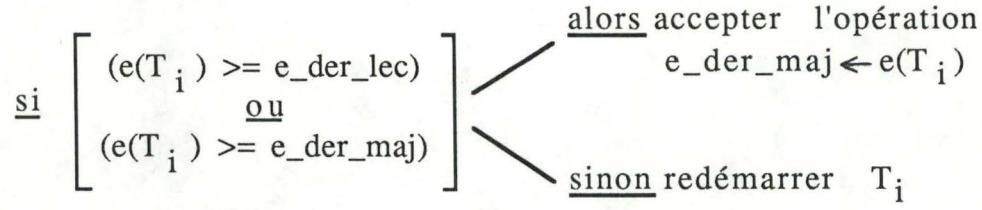
Le scheduler devra alors, pour une transaction T_i d'estampille $e(T_i)$, procéder comme suit :

◊ si T_i demande une **lecture**,

si $e(T_i) >= e\_der\_maj$
- alors accepter l'opération
  e_der_lec ← max ($e(T_i)$, e_der_lec)
- sinon redémarrer T_i

[1] Il devra donc y avoir atomicité du COMMIT.
[2] Une opération ne "réussit" que si elle a passé le stade du COMMIT.

◊ si T_i demande une **écriture**,



Outre ces informations, comme le scheduler doit obtenir confirmation du DM, il devra aussi, pour chaque objet, maintenir deux variables lui indiquant le nombre d'opérations de lectures et d'écritures non encore confirmées par le DM, de même qu'une file d'attente des opérations qui peuvent légalement (selon le protocole TO de base) s'attendre à cette confirmation.

La gestion de cet ensemble de données deviendra d'autant plus important que la taille de la BD accédée augmentera.

Tout comme nous avions accompagné la présentation du 2PL de ses deux versions, la première agressive, la seconde stricte, nous le ferons pour le TO.

4.3. L'estampillage strict

Tel qu'exposé ci-dessus, le TO de base produit des historiques sérialisables. Toutefois, rien ne garantit qu'ils seront aussi recouvrables, c'est-à-dire que le recouvrement pourra se faire complètement et de manière correcte.

L'exemple suivant en donne la preuve :

soit deux transactions telles que

T_1 : TR_BEGIN_1
$TR\_WRITE_1\ (X, V_{11})$
TR_COMMIT_1

T_2 : TR_BEGIN_2
$TR\_READ_2\ (X)$
$TR\_WRITE_2\ (Y, V_{21})$
TR_COMMIT_2

et

$e(T_1) < e(T_2)$

et supposons qu'un scheduler ait produit l'historique sérialisable suivant :

$W_1[X]\ R_2[X]\ W_2[Y]\ COMMIT_2 \parallel COMMIT_1$

Il faut se rendre à l'évidence que cet algorithme n'est pas recouvrable car T_2 a lu une donnée modifiée par T_1, a effectué certaines mises à jour et a été confirmée par le $COMMIT_2$. Or, T_1 n'a pas encore été confirmée à ce moment-là (avant le //). Si T_1 devait être défaite, il faudrait aussi pouvoir défaire T_2 qui en dépend, ce qui n'est plus possible.

La version stricte du TO est la version de base qui a été modifiée en conséquence. Nous avions signalé que cette dernière devait notamment maintenir à jour deux variables indiquant le nombre d'opérations de lecture et d'écriture non encore confirmées par le DM. C'est sur la seconde que notre version stricte devra jouer. Nous ne rentrerons pas dans les détails de cette modification; le lecteur pourra s'il le désire se référer à [BERN87].

4.4. L'estampillage conservateur

L'estampillage est par essence agressive. Lorsque l'ordre dans lequel le scheduler TO de base reçoit les opérations est fortement différent de celui des estampilles, il risque de rejeter trop d'opérations et par la même occasion, de provoquer l'échec fréquent de transactions. C'est pourquoi il est parfois préférable d'avoir à sa disposition une version plus conservatrice[1] du TO. C'est celle-ci que nous résumons ci-dessous.

La première approche est de retarder artificiellement (au niveau du scheduler) d'un certain délai les opérations provenant des transactions.

On espère par là qu'une transaction avec une estampille plus ancienne qui aurait éventuellement pu ne pas être terminée s'exécute au lieu d'être défaite suite à l'arrivée d'une transaction conflictuelle plus jeune.

Si nous reprenons l'exemple de la figure 4.1, nous pouvons le transformer de la manière suivante (fig. 4.2):

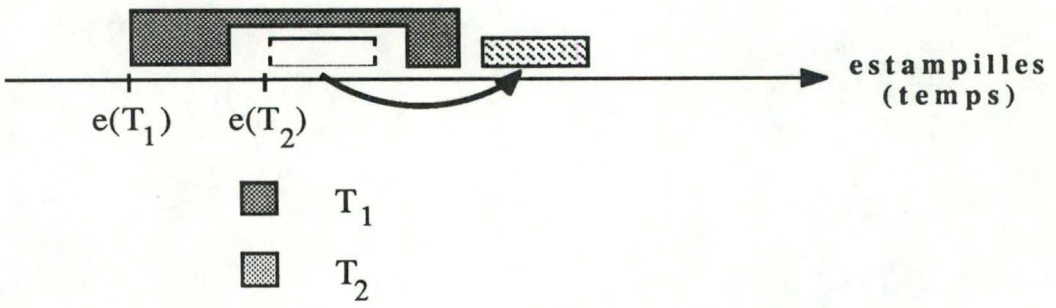


Figure 4.2 : Transaction plus jeune mise en attente

[1] Rappelons que "conservateur" signifie que le scheduler ne déclenchera les opérations que s'il est certain de ne provoquer aucun rejet (ABORT) de la transaction émettrice, ou du moins un minimum de rejets.

Si T_2 avait été exécutée selon la méthode de base, elle aurait provoqué l'avortement de T_1. Avec cette version retardée, T_1 a eu le temps de s'exécuter avant que T_2 n'exécute ses opérations, qui ne sont maintenant plus conflictuelles.

Au plus longtemps les opérations sont ainsi retardées, au plus on évite les risques de voir ce type d'avortements. L'inconvénient qui en résulte est bien sûr l'allongement des délais de traitement des transactions. Il s'agira de minimiser à la fois ces délais et le nombre de rejets, ces objectifs étant malheureusement contradictoires.

La seconde approche est elle tout à fait conservatrice puisqu'elle a pour but de ne jamais rejeter les transactions. Cependant, cela ne se fait pas sans imposer certaines contraintes. La plus usuelle est d'exiger que chaque transaction prédéclare les données auxquelles elle compte accéder. C'est la même solution que nous avions déjà envisagée pour le 2PL conservateur, et les inconvénients sont identiques. Nous ne détaillerons donc pas ce principe.

Il existe une forme plus complexe d'estampillage qui est celle de l'estampillage multi-dimensionnel présentée par Pei-Jyun et Bharat [PEIJ87]. Alors qu'un protocole d'estampillage traditionnel attribue une estampille prématurée à chaque transaction afin d'assurer un ordre sérialisable, quite à défaire des transactions parce que leur ordre d'apparition n'est pas conforme à celui recherché, dans un protocole à base d'estampillage multidimensionnel, chaque transaction se voit assigner un vecteur d'estampilles qui est dynamiquement mis à jour chaque fois qu'une de ses opérations arrive. Le scheduler se sert alors des informations de plus en plus précises concernant une transaction pour effectuer son contrôle de concurrence. Les auteurs distinguent plusieurs degrés de concurrence et étudient la complexité de la tâche prévisionnelle lorsque le degré de concurrence désiré augmente. Parmi les paramètres influençant la qualité du protocole, nous retiendrons le nombre d'éléments des vecteurs d'estampilles, considération qui doit être modérée par le coût de la gestion que la taille des vecteurs entraînera.

Chapitre 5

Le test du graphe de sérialisation : SGT

Nous avions déjà évoqué le nom de graphe de sérialisation dans le chapitre consacré à la sérialisatisabilité (chapitre 2). Le moment est venu de définir ce concept car il nous servira pour une technique particulière qui teste ce type de graphe, et qui porte à juste titre le nom de **SGT** (Serialisation Graph Testing).

5.1. Le graphe de sérialisation

Le **graphe de sérialisation** d'un historique H (pas nécessairement sérial !) portant sur un ensemble E de n transactions est un graphe orienté

◊ dont les noeuds sont les transactions de E qui sont confirmées (par un COMMIT) dans H

◊ et dans lequel chaque arc de T_i vers T_j signifie qu'une des opérations de T_i précède une opération conflictuelle[1] de T_j dans H.

Prenons par exemple les trois transactions suivantes :

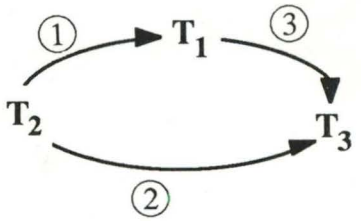
T_1	T_2	T_3
TR_BEGIN_1	TR_BEGIN_2	TR_BEGIN_3
$TR\_READ_1\ (X)$	$TR\_READ_2\ (X)$	$TR\_READ_3\ (X)$
$TR\_WRITE_1\ (X, V_{11})$	$TR\_WRITE_2\ (Y, V_{21})$	$TR\_WRITE_3\ (X, V_{31})$
$TR\_WRITE_1\ (Y, V_{12})$	TR_COMMIT_2	TR_COMMIT_3
TR_COMMIT_1		

et supposons qu'elles donnent lieu à l'historique ainsi constitué :

$R_1[X]\ R_2[X]\ W_1[X]\ R_3[X]\ W_3[X]\ W_2[Y]\ W_1[Y]\ TR\_COMMIT_{3,\ 2\ et\ 1}$

alors, nous obtenons le graphe de sérialisation :

[1] Deux opérations sont en conflit si une des deux au moins est une opération d'écriture.



L'arc ① provient du $R_2[X]$ qui précède le $W_1[X]$ (ou du $W_2[Y]$ avant le $W_1[Y]$), l'arc ② du R[X] précédant le W[X] et l'arc ③ du $W_1[X]$ devant le $R_3[X]$.

Notons qu'il n'y aura qu'un seul arc d'une orientation donnée entre deux transactions, même si plusieurs opérations sont en conflit dans le même sens.

Puisqu'un arc de T_i vers T_j signifie que selon cet historique H, une opération de T_i est en conflit et précède une opération de T_j, ce type de graphe suggère que T_i devrait précéder T_j dans tout historique sérial qui serait équivalent à H.

Un théorème, dit **théorème de la sérialisabilité**, affirme en effet que s'il n'y a pas de cycles dans le graphe de sérialisation d'un historique H, alors il est possible de trouver un historique sérial équivalent à H, ce qui prouve donc que H est sérialisable.

Rechercher un tel équivalent de H revient à essayer de trouver un historique cohérent avec le graphe de sérialisation de H. Dans l'exemple donné ci-dessus, T2 T1 T3 est un historique sérial (c'est même le seul que nous puissions trouver dans ce cas).

Les schedulers SGT que nous nous proposons de présenter auront par conséquent pour objectif d'empêcher la formation de cycles dans les graphes de sérialisation.

Remarque :

Les graphes utilisés par les méthodes à base de SGT diffèrent quelque peu des graphes de sérialisation tels que la théorie les utilise et tels que nous venons de les définir.

En effet, alors que le graphe théorique ne contient que des noeuds repésentant toutes les transactions confirmées et seulement celles-là, le graphe utilisé dans la pratique

- ◊ *pourra ne pas inclure toutes les transactions qui ont été confirmées, notamment celles qui l'ont été loin dans le passé;*
- ◊ *inclura généralement des noeuds pour toutes les transactions actives, à savoir celles qui par définition n'ont pas encore été confirmées.*

Il est nécessaire de modifier ainsi le graphe théorique car les concepts qu'il représente sont très intéressants mais inutiles s'ils ne font que témoigner de l'état passé des faits.

Nous développerons deux versions du SGT : la version de base et la version conservatrice. Par abus de langage, lorsque nous utiliserons le mot "graphe", sauf mention contraire, cela signifiera "graphe de sérialisation".

5.2. Protocole de base

Le scheduler **SGT de base** respecte le protocole suivant :

◊ lorsqu'il reçoit une demande quelconque $D_i[X]$ d'une transaction T_i, il ajoute un noeud pour T_i dans son graphe - à moins qu'il n'existe déjà - et ensuite trace un arc de T_j vers T_i pour chaque opération $D_j[X]$ traitée plus tôt et en conflit avec $D_i[X]$;

◊ ensuite, le scheduler décidera de la première ou de la seconde action, selon que le graphe est devenu cyclique ou non :

- si le graphe est devenu cyclique : si $D_i[X]$ était acceptée et traitée dans l'état actuel des choses, il en résulterait une exécution non-sériale. Le scheduler doit donc rejeter $D_i[X]$, avorter T_i (en envoyant une demande d'ABORT de T_i au DM) et, ceci fait, retirer du graphe le noeud correspondant à T_i et tous les arcs qui y mènent[1];

- si le graphe reste non-cyclique : le scheduler peut alors propager la demande de T_i vers le DM à condition toutefois que toutes les opérations conflictuelles avec $D_i[X]$ soient déjà terminées. Sinon, le scheduler doit retarder l'opération jusqu'à ce qu'il reçoive confirmation du DM.

Ce protocole utilisera généralement le principe de l'estampillage de base (voir supra le point qui était consacré au TO de base), méthode qui, nous l'avons vu, résolvait le problème du respect de l'ordre parmi un historique d'opérations conflictuelles.

Les griefs que l'on peut trouver à ce protocole sont tout à fait semblables à ceux décelés pour le TO de base, à savoir principalement que, pour chaque objet X, il est nécessaire de pouvoir retrouver les opérations que toutes les transactions actives ont pu effectuer dessus (par la gestion d'un certain nombre d'ensembles et de compteurs), mais ils sont encore accentués par deux considérations propres au SGT :

◊ le maintien des graphes de sérialisation et la recherche de cycles dans ceux-ci constitue une lourde surcharge de travail;

◊ la place occupée par les ensembles et compteurs, encore plus volumineuse que pour le TO, ne peut pas être libérée dès que la transaction qui la requiert se termine.

[1] Le graphe est redevenu exempt de cycles, puisque le "coupable" a été défait. L'exécution produite par le scheduler est donc à nouveau sériale.

Supposons par exemple une transaction T_1 ayant formulé la demande $R_1[X]$, puis une série d'opérations $O_i[]$ (respectant le protocole SGT) pour des transactions T_i ($i > 1$), portant sur des objets Y_i (Y_i tous différents de X) et qui ont été confirmées. Si T_1 veut maintenant effectuer un $W_1[X]$, le scheduler doit être capable de dire si cet objet X est différent de tous les Y_i qui ont été accédés par les transactions T, même si celles-ci ont déjà été confirmées.

En réalité, le scheduler ne pourra se débarasser des informations concernant une transaction T_i que s'il est certain que T_i ne sera plus jamais impliquée dans un cycle à l'avenir. C'est-à-dire s'il n'existe plus d'arcs arrivant au noeud i puisque de tels arcs ne peuvent plus surgir une fois que la transaction a été confirmée.

Nous pouvons donc conclure que cette technique permet de traiter tous les mélanges d'opérations, alors que le 2PL n'en autorisait que certains et que le TO n'admettait que ceux dans l'ordre des estampilles, mais elle y arrive moyennant un excès de travail considérable. Cependant, étant donné qu'il s'agit d'une technique assez récente, nous ne dispose à l'heure actuelle que de peu de renseignements quant à ses performances.

5.3. Le SGT conservateur

Comme nous l'avons déjà mentionné pour les autres techniques, lorsque le scheduler est dit conservateur, cela signifie qu'il ne rejette jamais les opérations mais qu'il les retarde plutôt. La solution est dès lors la même, à savoir la prédéclaration.

Tout comme pour le protocole de base, la version conservatrice du SGT maintient à jour un graphe de sérialisation des transactions. Pour chaque nouvelle transaction, le scheduler crée un nouveau noeud dans le graphe et pour chaque opération spécifiée grâce aux ensembles prédéclarés, insère un arc conformément au protocole de base.
La différence se situe au niveau de la gestion de la file d'attente des opérations retardées.

Pour chaque file d'attente attachée à un objet X, le scheduler s'arrange pour que :

- ◊ les opérations non conflictuelles soient stockées dans un ordre arbitraire, soit par exemple l'ordre d'arrivée;
- ◊ les opérations conflictuelles soient stockées dans un ordre conforme au graphe, c'est-à-dire que s'il y a un arc de T_j vers T_i dans le graphe, alors l'opération $O_j[X]$ sera mise dans la file d'attente de X avant $O_i[X]$, c'est-à-dire que $O_j[X]$ sera finalement traitée avant $O_i[X]$.

Lorsque le scheduler peut faire exécuter une nouvelle opération par le DM, il recherche dans une des files d'attente si une opération est "prête".

Une opération $O_i[X]$ sera dite **prête** lorsque :

◊ les exécutions de toutes les opérations déjà transmises au DM et en conflit avec $O_i[X]$ seront confirmées en tant que telles par le DM;

◊ pour chaque transaction T_j précédant immédiatement T_i dans le graphe, et pour chaque ensemble prédéclaré E_j d'un type d'opération (lecture ou écriture) en conflit avec $O_i[X]$,

- soit X n'appartient pas à E_j,

- soit l'opération en question a déjà été reçue par le scheduler.

La première condition est une convention classique garantissant que les opérations conflictuelles soient bien exécutées dans l'ordre approprié par le DM.

La seconde condition est l'essence-même de cette méthode puisqu'elle est destinée à éviter les avortements. Elle a pour but de retarder l'exécution d'opérations qui pourraient encore être suivies d'autres opérations conflictuelles. Le fait qu'elle ne s'applique qu'aux transactions qui précèdent immédiatement T_i vient de ce que la condition est nécessairement satisfaite par les autres transactions.

Le scheduler répètera ce processus jusqu'à ce que chaque file d'attente soit vide ou jusqu'à ce qu'il n'y ait plus d'opérations prêtes.

Les remarques concernant les avantages et inconvénients de cette méthode sont les mêmes que pour le protocole de base. Nous ne nous y attarderons pas, si ce n'est pour noter que la surcharge de travail sera encore plus lourde puisque la version conservatrice doit aussi éviter les avortements. En outre, l'exigence de la prédéclaration n'est certainement pas faite pour favoriser cette méthode.

Chapitre 6

Les méthodes optimistes

Lors de la présentation des techniques de verrouillage et d'estampillage, nous avons vu que l'objectif était de prévenir les conflits. Cependant, si nous envisageons certaines situations où les conflits sont presque inexistants ou du moins très rares, alors le coût engendré par l'une ou l'autre de ces techniques peut se révéler excessif. De là l'idée de n'effectuer aucun contrôle particulier a priori et de guérir les quelques cas "malades" lorsqu'ils se présenteront.

De telles méthodes peuvent être considérées comme optimistes dans la mesure où elles espèrent que les conflits ne se produiront pas. Si toutefois elles apprennent l'existence d'un conflit, alors elles comptent sur la possibilité de défaire des transactions pour réparer les dégâts.

Au plus les conflits seront rares, au plus ces méthodes deviendront efficaces et performantes aux dépens des méthodes de verrouillage ou d'estampillage. Nous essaierons donc de déterminer le profil des applications présentant une telle particularité.

6.1. Principe général

Jusqu'à présent, lorsque le scheduler recevait les demandes des transactions, il devait décider soit d'accepter, soit de retarder, soit de refuser de traiter ces opérations.

L'approche des méthodes optimistes est de les accepter immédiatement, mais de vérifier de temps en temps si des opérations conflictuelles n'ont pas été acceptées trop hâtivement. Le scheduler devra alors avorter certaines transactions. C'est donc l'attitude agressive poussée à l'extrême.

Les **méthodes optimistes** reposent sur une découpe en deux ou trois phases de toute transaction, découpe dont la validation est le noyau, comme l'indique la figure 6.1 :

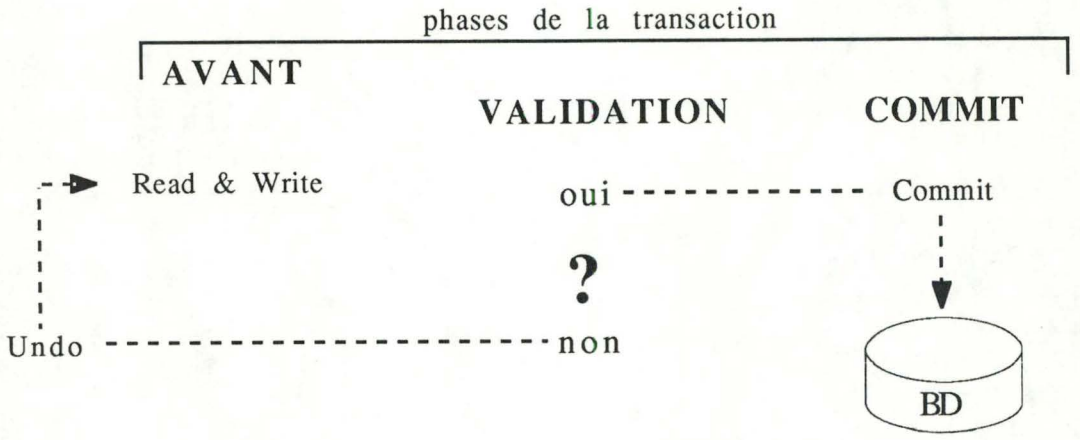


Figure 6.1 : Découpe en 2 ou 3 phases d'une transaction

Toute transaction sera découpée en :

◊ une phase avant, pendant laquelle toutes les opérations peuvent avoir lieu sans restriction aucune;

◊ une phase de validation, qui servira à établir que les mises à jour apportées lors de la phase avant n'affecteront pas l'intégrité de la BD;

◊ et, éventuellement, une phase de commit, durant laquelle les modifications seront définitivement confirmées.

Le scheduler passera à la phase de validation juste avant de traiter le COMMIT de chaque transaction. Il vérifiera alors si l'exécution comprenant le COMMIT en question est sérialisable. Si elle ne l'est pas, il devra avorter cette transaction.

Cette validation est obligatoire pour les transactions rédactrices, pas pour les transactions lectrices à moins que les données qu'elles ont lues ne servent à d'autres transactions rédactrices.

Remarquons encore que cette validation, parfois aussi appelée ***certification****, doit au moins avoir lieu à chaque COMMIT d'une transaction; le scheduler (ou "****certifieur****") risquerait en effet d'acter définitivement les modifications d'une transaction non sériale dans la BD.*

Ces certifieurs pourront être basés sur chacune des trois méthodes que nous avons vues dans cette première partie : le verrouillage à deux phases (2PL), l'estampillage (TO) et le test du graphe de sérialisation (SGT). Nous verrons ci-dessous comment ces certifieurs fonctionnent.

6.2. La certification à base de 2PL

Lorsque le certifieur reçoit une demande d'une transaction T_i, il enregistre certaines données concernant cette demande, puis la transmet sans autre mesure au DM.

Les renseignements gardés par le certifieur sont, pour chaque transaction, l'ensemble des données qu'elle a demandé de lire et l'ensemble des données qu'elle a demandé de mettre à jour.

Ceux-ci permettront au certifieur, lorsqu'il recevra le COMMIT de T_i, de vérifier si une quelconque des opérations de T_i qu'il a transmises au DM était en conflit avec une des autres transactions actives[1] T_j. Dans l'affirmative, T_i devra être défaite; dans le cas contraire, la demande de COMMIT de T_i pourra être transmise au DM, confirmant ainsi la réussite de T_i (c'est-à-dire que le certifieur passe alors à la phase d'écriture de la transaction, cfr. figure 6.1).

Remarque :

Il est inutile de chercher dans cette certification la présence réelle de la règle des deux phases ou d'un quelconque verrouillage. Le nom de certification 2PL vient sans doute du fait que, tout comme le scheduler 2PL rejète systématiquement une opération conflictuelle, le certifieur 2PL refusera une certification s'il se rend compte avoir fait exécuter des opérations conflictuelles. Cette similitude est confirmée par le fait qu'un historique sérialisable produit par l'un aurait tout aussi bien pu être produit par l'autre.

Il serait aussi possible de modifier ce certifieur pour le rendre, par exemple, plus imperméable aux avortements en cascade (puisqu'il faut aussi défaire les transactions dépendant de celle à qui on a refusé la certification). Cela irait toutefois à l'encontre de l'esprit optimiste de ces certifieurs.

6.3. La certification à base de TO

Le principe du certifieur à base d'estampillage n'est guère différent de celui présenté ci-dessus : il transmet évidemment les opérations dès qu'il les reçoit des transactions et effectue la validation au moment où il reçoit la demande de COMMIT.

Ce qui différencie le certifieur TO du certifieur 2PL c'est la condition de rejet d'une transaction. En effet, une transaction pourra être confirmée même si on avait accepté pour elle des opérations en conflit avec celles d'autres transactions, pourvu que tous ces conflits respectent l'ordre des estampilles. Il s'agit donc effectivement d'une certification à base d'estampillage.

---

[1] Cela suppose donc que le certifieur tienne un journal des transactions dites "actives", c'est-à-dire celles qui n'ont pas encore été confirmées (par un COMMIT).

Remarque :

Alors que le scheduler TO rejetait la transaction dès que l'ordre des estampilles était rompu, le certifieur ne le fait qu'au moment de la certification.

Nous ne détaillerons pas ce type de certifieur outre mesure car, étant donné la remarque qui vient d'être faite, le scheduler s'annonce préférable au certifieur puisque ce dernier continue à transmettre les opérations de la transaction au DM même lorsqu'il n'y a plus aucun espoir pour cette transaction de réussir son COMMIT (alors que dans pareille situation, le scheduler aurait déjà cessé de le faire).

Signalons simplement que Kung et Robinson proposent dans [KUNG81] deux familles de méthodes optimistes basées sur le principe de l'estampillage. Ils discutent du moment le plus propice à l'attribution de l'estampille pour une transaction, car l'issue de cette décision influence le degré de parallélisme qui pourra en être obtenu. Ils prouvent également la validité de leurs propositions.

6.4. La certification à base de SGT

La dernière méthode que nous pouvons tenter d'appliquer aux certifieurs est celle des tests de graphes de sérialisation.

Tout comme pour la version de base de la méthode SGT, le certifieur gère en permanence un graphe de sérialisation des transactions passées et présentes.

Chaque fois que le certifieur reçoit une opération $O_i[X]$ d'une transaction T_i, il ajoute dans le graphe un arc de T_j vers T_i s'il a déjà envoyé une opération en conflit avec $O_i[X]$ au DM. Ensuite, peu importe si ce graphe contient un cycle, il transmet la demande $O_i[X]$ au DM[1].

Le principe de certification peut maintenant entrer en vigueur : lorsque le certifieur reçoit le COMMIT d'une transaction, il vérifie si celle-ci a été impliquée dans un cycle. Si oui, il avorte cette transaction; sinon, il la certifie en demandant au DM de la confirmer.

Par définition, cette technique héritera des avantages et inconvénients et des certifieurs et du SGT. Les mêmes problèmes de place mémoire apparaissent et peuvent être résolus partiellement par les mêmes procédés.

Nous retiendrons que le test du graphe de sérialisation (SGT) est de nature à s'appliquer aux techniques de certification puisqu'il mettait déjà en oeuvre l'outil de base de la certification.

---

[1] Il convient bien sûr de s'assurer que les opérations sont bien traitées dans l'ordre de la transmission vers le DM.

Chapitre 7

Les données à versions multiples

Jusqu'à présent, nous avions implicitement supposé que les données n'existaient qu'en une seule version et que chaque mise à jour se faisait sur cet exemplaire unique. Il est cependant des situations où il peut être intéressant de ne pas "écraser" cette valeur mais au contraire de la garder. A chaque modification, une nouvelle copie, appelée **version**, d'une donnée est créée, et pour chaque lecture, il est possible de spécifier quelle version d'un objet est demandée.

Si nous avons considéré le problème de la gestion des versions multiples d'un objet dans le cadre de ce travail, c'est pour la raison suivante : alors que dans un système à version unique, une lecture peut être rejetée parce qu'elle arrive trop tard - elle était censée lire une donnée qui vient d'être écrasée -, dans un système à versions multiples, le scheduler a la possibilité de satisfaire cette demande en lui offrant une version plus ancienne de l'objet.

Les techniques à versions multiples sont donc un moyen d'augmenter la concurrence d'un système, mais il convient de modérer quelque peu ces propos par les remarques suivantes :

Nous avions déjà signalé (cfr. point 3.3.3.) que le recouvrement utilisait certains mécanismes dont le contrôle de concurrence pouvait se servir, moyennant quelques adaptations. Les "shadow values" en étaient un exemple. Certaines techniques de recouvrement utilisent un autre principe, celui des "images avant". Sommairement, il s'agit d'enregistrer, avant la modification d'un objet, son état afin de pouvoir le restaurer en cas de panne du système ou d'avortement d'une transaction.

Il peut donc être assez facile de modifier la structure du DM de façon à ce qu'il conserve ces "images avant" à la disposition du scheduler, auquel cas le surcoût de gestion des versions n'est pas excessif.

Cependant, la multiplication des copies d'une donnée accroissant sensiblement l'espace de stockage nécessaire, il sera indispensable de mettre (régulièrement) de l'ordre dans ces versions, par exemple en les archivant ou en les détruisant.

Dans la suite du travail, nous supposerons que le mot "version" désigne une copie de donnée produite soit par une transaction active soit par une transaction confirmée (et par conséquent, terminée). Une transaction avortée ne laisse derrière elle aucune copie qu'elle aurait pu créer. De plus, si une transaction T_1 lit une version produite par une transaction T_2 encore active, le scheduler doit obliger T_1 à attendre que T_2 se soit terminée; si c'est par

un COMMIT, aucune mesure particulière concernant T_1 ne doit être prise, mais si c'est par un ABORT, alors T_1 doit aussi être avortée.

Nous poserons également pour hypothèse que les versions multiples n'ont pour objectif que d'aider à augmenter la concurrence et non pas de satisfaire des besoins d'utilisateurs qui auraient explicitement accès aux différentes versions d'un objet. L'existence de cette multiplicité est donc transparente aux utilisateurs qui n'ont conscience d'accéder qu'à un seul objet. C'est l'essence-même des protocoles présentés ci-dessous que d'assurer cette transparence de la multiplicité des copies d'une donnée.

7.1. Verrouillage à deux phases multiversion

Dans le cadre du verrouillage basé sur le 2PL pour objets à versions multiples, nous commencerons par présenter le protocole lorsqu'il n'y a que deux versions d'un objet. Nous le généraliserons ensuite au cas des versions multiples (plus de deux).

7.1.1. 2PL pour deux versions : 2V2PL

En 2PL de base (pour une seule copie), un verrou X sur un objet empêche toute opération d'une autre transaction sur cet objet. Ce protocole un peu trop restrictif peut être modifié de sorte à accroître le parallélisme potentiel.

En effet, supposons qu'un objet puisse exister en deux versions. Lorsqu'une transaction T_i voudra modifier un objet Y, elle créera une nouvelle version de Y, notée Y_i et verrouillera Y de façon à empêcher les autres transactions et de lire Y_i et de créer une nouvelle version de Y. Les lectures peuvent cependant être satisfaites puisqu'il suffit de leur offrir l'ancienne valeur de Y, celle créée par la dernière transaction qui ait été confirmée avant T_i.

La gestion de ces deux versions est laissée aux soins du DM; nous n'aborderons pas les techniques qu'il peut utiliser pour parvenir à effectuer sa tâche.

Si nous notons les versions d'un objet Y par Y_i, Y_j, $Y_{...}$ ... où l'indice identifie la transaction qui a créé cette version, une opération d'une transaction $\underline{k}$ sur un objet Y sera toujours[1] de la forme :

◊ $W_k[Y_k]$ pour une écriture, ou

◊ $R_k[Y_i]$ pour une lecture.

[1] Du point de vue du scheduler, car du point de vue de l'utilisateur, nous supposons qu'il y a transparence de la multiversion.

Le protocole **2V2PL** (Two Version Two Phase Locking) nécessite trois types de verrous : de lecture, d'écriture et de validation. Les deux premiers (R et W) sont utilisés de la même manière qu'en 2PL "mono-copie". Le dernier type de verrou, V (pour Validation) est gouverné par des besoins propres au 2V2PL et nous en expliquerons l'usage lors de la présentation du protocole, suite au tableau 7.1.

Le tableau 7.1 ci -dessous est l'adaptation du tableau 3.1 pour le 2V2PL, c'est-à-dire en tenant compte du nouveau type de verrou introduit, celui de validation :

Tableau 7.1 : Compatibilité des verrous en 2V2PL

Compatibilité ?		Verrou posé en		
		Lecture	Ecriture	Validation
Verrou demandé en	Lecture	OUI	OUI	NON
	Ecriture	OUI	NON	NON
	Validation	NON	NON	NON

En comparant les tableaux 3.1 et 7.1, nous constatons que l'introduction d'une nouvelle version pour un objet et du protocole 2V2PL atteint bien son objectif puisque nous retrouvons dans ce tableau 7.1 deux cas pour lesquels il n'y a plus le conflit trouvé au tableau 3.1 (il s'agit de lecture/écriture et inversément).

Pour une transaction T_i voulant accéder à un objet Y, le **protocole 2V2PL** est alors le suivant :

◊ lorsque le scheduler reçoit une demande d'écriture, soit $W_i[Y]$, il tente de poser un verrou de type $WL_i[Y]$. En consultant le tableau des compatibilités (tab. 7.1), nous déduisons qu'il doit retarder cette opération jusqu'à ce que tous les verrous de type W ou V sur Y d'autres transactions aient été relâchés; sinon, il pose le verrou $WL_i[Y]$ et transforme de W[Y] en W[Y] qu'il transmet au DM;

◊ lorsque le scheduler reçoit une demande de lecture, soit $R_i[Y]$, il tente de poser un verrou de type $RL_i[Y]$. Puisque les verrous de lecture n'entrent en conflit qu'avec les verrous de type V (cfr. tableau 7.1),

- si T_i détenait déjà un verrou $WL_i[Y]$ - ce qui suppose qu'elle a déjà écrit dans Y_i -, le scheduler transforme le R[Y] en R[Y] qu'il envoie au DM;

- sinon, il doit attendre que de pouvoir poser un verrou R sur Y, c'est-à-dire attendre que tous les verrous V d'autres transactions sur Y aient été libérés, puis pose le verrou R et transforme le $R_i[Y]$ en $R_i[Y_j]$ qu'il envoie au DM;

◊ enfin, lorsque le scheduler reçoit une demande de confirmation, soit $COMMIT_i$, il tente de convertir les verrous W_i de T_i en verrous V_i. Puisque les verrous V sont en conflit avec les R, le scheduler ne peut réaliser cette "conversion" que lorsqu'il est certain qu'il n'y a plus de verrous de lecture sur Y; sinon, il doit attendre la libération de tous ces verrous.

Remarques :

Nous avons, dans le traitement du COMMIT, supposé disposer d'une conversion de verrous de type W->V. Ainsi que nous l'avions déjà expliqué lors de la sous-section consacrée aux interblocages en 2PL (sous-section 3.5.2.), cette conversion de verrous risque de provoquer des interblocages. Par exemple, si T_1 détient un $RL_1[Y]$ et T_2 un $WL_2[Y]$, l'interblocage se produira si T_1 essaie de convertir son verrou en $WL_1[Y]$ et T_2 le sien en $VL_2[Y]$. Une des méthodes classiques de détection ou de prévention peut alors être utilisée pour y remédier.

7.1.2. 2PL pour plus de deux versions : MV2PL

Dans le protocole 2V2PL, le but des verrous d'écriture est simplement d'assurer qu'il n'existe que deux versions d'un objet à un moment donné. Affaiblir cette condition risque de créer plusieurs versions "non-validées"[1] d'un objet. Néanmoins, si nous respectons les autres règles du 2V2PL, alors seule la dernière version validée pourra être lue. Autoriser l'accès à des versions non-encore validées risque cependant d'amener des avortements en cascade[2].

Le **protocole MV2PL** (MultiVersion 2PL) est une adaptation du 2V2PL qui cherche à éviter les avortements en cascade tout en permettant l'existance de plusieurs versions non-validées.

Le scheduler 2V2PL a dû subir deux modifications avant de donner lieu au scheduler MV2PL. Les voici :

◊ une transaction ne peut être validée avant que toutes les versions qu'elle a lues n'aient été validées, et,

◊ le scheduler ne peut effectuer la conversion W->V que s'il n'y a plus de verrous de lecture sur des versions validées d'un objet.

---

[1] Les versions "non-validées" d'un objet sont celles créées par des transactions non encore confirmées.

[2] Nous avions déjà fait remarquer que lorsque des transactions lisaient des données modifiées par d'autres transactions, en cas d'avortement de ces dernières, il fallait aussi avorter toutes celles qui s'étaient basées sur leurs "inputs". C'est ce qu'on appelle une "cascade d'avortements".

Ces deux règles ont pour objectif d'ignorer un verrou de lecture posé sur une version non-validée d'un objet jusqu'à ce que, soit cette version est validée, soit la transaction ayant posé le verrou demande à être elle-même validée, et par la même occasion d'éviter la cascade d'avortements tout en offrant une concurrence accrue. Le coût de cette gestion a déjà été discuté en introduction à ces méthodes multiversions.

7.2. Estampillage multiversion : MVTO

Il est évidemment possible de construire un scheduler gérant les données à versions multiples basées sur l'estampillage. Chaque transaction est identifiée par une estampille (qui accompagne les opérations qu'elle effectue) et chaque version d'un objet est par conséquent marquée de l'estampille de la transaction qui l'a créée.

Le **protocole MVTO** (MultiVersion Timestamp Ordering) est aisé à comprendre. Son objectif est de transformer les opérations en provenance des transactions sur des objets en opérations sur des versions d'objets tout en assurant la transparence de la multiversion. Les transactions devront donc être perçues comme gérées par un scheduler TO pour BD à exemplaires uniques.

En supposant que le scheduler traite les demandes des transactions dans leur ordre d'arrivée, le protocole est le suivant :

◊ lorsque le scheduler reçoit d'une transaction T_i une demande de lecture, soit $R_i[Y]$, il la transforme en $R_i[Y_k]$, où Y_k est "la version de Y avec la plus grande estampille inférieure ou égale à celle de T_i", et envoie $R_i[Y_k]$ au DM;

◊ lorsque le scheduler reçoit d'une transaction T_i une demande d'écriture, soit $W_i[Y]$, il vérifie s'il a déjà traité une demande $R_j[Y_k]$ telle que : $e(T_k) < e(T_i) < e(T_j)$;

- si oui, il rejette la demande $W_i[Y]$;
- sinon, il transforme $W_i[Y]$ en $W_i[Y_i]$ et l'envoie au DM;

◊ enfin, il ne traitera une demande de confirmation (COMMIT) de T_i que s'il a traité toutes les demandes de COMMIT des transactions qui ont créé (ou modifié) des versions lues par T_i.

La règle de lecture est triviale puisque "la version de Y avec la plus grande estampille inférieure ou égale à celle de T_i" est bien la valeur qu'un scheduler TO pour BD à version unique aurait renvoyée.

La règle d'écriture l'est un peu moins; c'est pourquoi nous proposons l'éclaircissement suivant :

supposons que nous ayons trois transactions T1, T2 et T3 par ordre croissant d'estampilles, définies comme suit :

T_1	T_2	T_3
TR_BEGIN_1	TR_BEGIN_2	TR_BEGIN_3
$TR\_WRITE_1\ (X, V_1)$	$TR\_WRITE_2\ (X, V_2)$	$TR\_READ_3\ (X)$
TR_COMMIT_1	TR_COMMIT_2	TR_COMMIT_3

et supposons que le scheduler ait déjà envoyé au DM les instructions :

$W_1[X_1]\ R_3[X_1]$

Lorsqu'il reçoit le TR_WRITE de T_2, s'il le transforme en $W_2[X_2]$ et l'envoie au DM, il produit une exécution qui n'est plus celle qu'aurait produit un scheduler TO pour exemplaires uniques. En effet, T_3 aurait dû lire la valeur de X écrite par T_2, alors que l'exécution

$W_1[X_1]\ R_3[X_1]\ W_2[X_2]$

donne à lire à T_2 la valeur écrite par T_1. Afin d'éviter ce problème, le scheduler rejète le TR_WRITE_2. Cet exemple est illustré à la figure 7.1 ci-dessous.

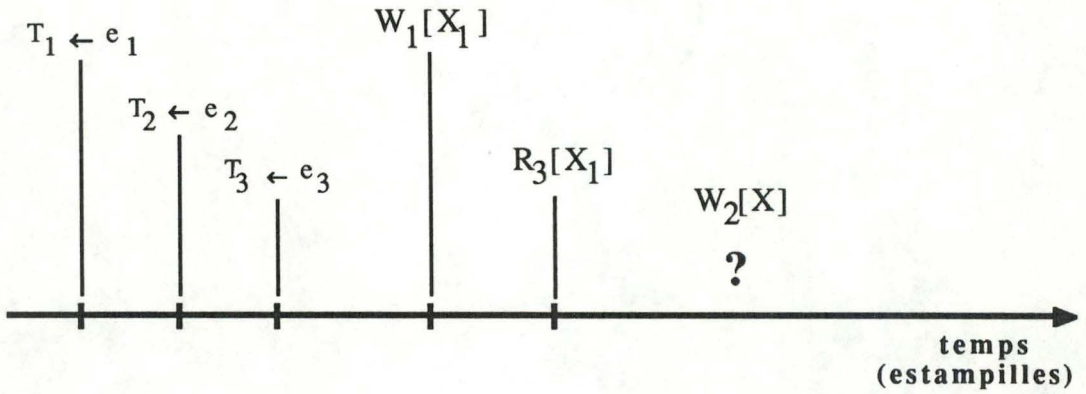


Figure 7.1 : Justification de la règle d'écriture en MVTO

C'est ce qui explique que, de façon générale, le scheduler rejètera une demande d'écriture i s'il a déjà traité une lecture j sur une version k telle que $e(T_k) < e(T_i) < e(T_j)$.

Trouver la version adéquate à renvoyer de manière à éviter l'"invalidation" des ordres de lecture est un leurs important de ce protocole. Le scheduler doit en effet conserver certaines informations quant aux opérations qu'il a déjà traitées.

En fait, pour chaque version Y_i d'un objet Y, il doit mémoriser des renseignements semblables à ceux que le scheduler TO de base utilisait (cfr. section 4.2), à savoir un intervalle $I(Y_i)$ = [e_maj, e_lec_max], où e_maj est l'estampille de Y_i et e_lec_max la plus grande estampille parmi toutes les transactions qui ont lu Y_i (s'il n'y en a pas, alors e_lec_max vaut e_maj).

Lorsqu'il reçoit un $\mathbf{R_k}[Y]$,

◊ le scheduler cherche dans l'ensemble des $I(Y_i)$ de Y afin d'y trouver le $I(Y_j)$ ayant le plus grand e_maj inférieur ou égal à $e(T_k)$, (1)

◊ si $e(T_k) > e\_lec\_max$, alors e_lec_max prend cette valeur $e(T_k)$,

◊ et envoie finalement l'ordre $R_k[Y_j]$ au DM.

Lorsqu'il reçoit un $\mathbf{W_k}[Y]$,

◊ effectue les mêmes opérations qu'en (1),

◊ si $e(T_k) < e\_lec\_max$, alors rejète $W_k[Y]$
sinon envoie $W_k[Y_k]$ au DM,
et effectue pour $I(Y_k)$: $e\_maj = e(T_k)$
$e\_lec\_max = e(T_k)$

Des problèmes d'espace mémoire ou d'espace disque risquent à nouveau de surgir, auquel cas scheduler et DM devront effectuer un nettoyage des versions, par ordre d'ancienneté.

Pour ces deux adaptations aux données à versions multiples (MV2PL et MVTO), une théorie de la sérialisabilité a dû être développée et de nouvelles preuves de validité apportées. Notre optique étant de ne pas nous attarder sur les aspects purement formels des algorithmes, nous préférons renvoyer le lecteur à une référence en matière de preuves de protocoles de contrôle de concurrence, [BERN87].

Notons encore que dans la même référence, Bernstein et al. proposent une méthode dite "mixte" de gestion de versions multiples. Les méthodes "mixtes", qui font l'objet d'un section future (section 8.3), sont appelées ainsi car elles intègrent deux méthodes basées sur des concepts différents pour résoudre des sous-problèmes distincts (ou presque) afin de profiter des avantages de chacune. La méthode mixte qu'ils proposent utilise le MVTO afin de règler les problèmes de lecture et le 2PL strict pour ceux de mise à jour. Il nous semble préférable d'en venir immédiatement aux conclusions qu'ils apportent à leur méthode : les lectures et les écritures ne se gênent pas mutuellement, mais peuvent se gêner entre elles, ce qui n'est pas le cas du 2PL de base, du 2V2PL et du MV2PL dans lesquels une consultation peut retarder une mise à jour; l'inconvénient majeur cependant , est que leur méthode mixte

offre à des lecteurs des versions parfois périmées et qu'elle implique une gestion considérable d'analyse d'estampilles. Suite à ces deux dernières remarques d'ailleurs, ils proposent deux compromis à base de gestion de listes de COMMITs, que nous ne développerons pas dans ce travail.

Citons enfin Carey et al. qui ont consacré un article [CARE86] afin de présenter des résultats comparatifs de différents algorithmes de contrôle de concurrence, les uns pour BD à exemplaires uniques, les autres lorsque les données peuvent exister en plusieurs versions (MVTO, MV2PL entre autres). Ils y concluent que tous les algorithmes multiversions donnent des résultats sensiblement meilleurs que leurs équivalents "mono-copies", que la surcharge de stockage destiné aux anciennes versions n'est pas toujours si volumineuse qu'on le croit et que le coût de gestion des versions peut être fortement réduit si le module qui en est responsable est imlpémenté efficacement.

Chapitre 8

Verrouillage vs non-verrouillage

Ce chapitre compare d'abord les techniques avec verrouillage et les techniques sans verrouillage (estampillage, test du graphe de sérialisation, certification et méthodes optimistes) : les atouts et les insuffisances majeurs de chaque technique sont envisagés. Est ensuite envisagée une technique visant à instaurer un compromis entre les deux types de techniques. Il s'agit d'une méthode particulière dans laquelle le problème du contrôle de concurrence est divisé en deux sous-problèmes résolus presque indépendamment par deux méthodes, éventuellement différentes.

Nous ne rappellerons pas les principes à la base des différentes techniques; le lecteur pourra, si nécessaire, se référer aux chapitres que nous venons de présenter dans cette première partie.

8.1. Avantages et inconvénients du verrouillage

Remarque :

Si nous comparons le verrouillage avec l'approche purement séquentielle, nous constatons évidemment que la gestion des verrous représente une surcharge de travail qui n'existait pas lorsqu'on interdisait le parallélisme. Cette surcharge vaut pour toutes les techniques, avec ou sans verrouillage, et ne consitue donc pas un argument probant en faveur d'une quelconque d'entre elles.

Le premier inconvénient des techniques de verrouillage est que même des transactions de simple consultation devront passer par ce mécanisme alors qu'elles ne peuvent affecter la cohérence de la BD et ceci afin de garantir que les données lues ne soient pas modifiées par d'autres transactions simultanées.

Le verrouillage implique généralement aussi des possibilités d'interblocage, qui devront être soit prévenues, soit détectées[1]. Ces tâches consommant des ressources, elles doivent également être portées au passif des techniques de verrouillage.

Si suite à un conflit, une transaction doit être défaite, alors ses verrous doivent rester posés jusqu'à ce que cette opération soit complètement terminée. C'est là encore un grief en défaveur du verrouillage.

Et enfin, ce qui constitue sans doute l'argument-clé des opposants du verrouillage : il n'est peut-être nécessaire que dans le pire des cas.

Cependant, et c'est là sans nulle conteste leur atout premier, elles sont fondées sur un principe simple. Cette simplicité n'est pas que conceptuelle, bien que ce soit déjà un aspect positif; elle est aussi pratique : à des concepts simples s'associent souvent des techniques simples. Une gestion de verrous n'a pas la lourdeur d'une certification, par exemple.

8.2. Avantages et inconvénients de l'estampillage

L'estampillage est une méthode qui procède, par allocation de priorités (ou estampilles) aux transactions, à un classement selon un ordre strictement croissant. Son intérêt, puisque ne procédant pas par verrouillage, est d'éviter tout interblocage. Cependant, sa nature agressive risque de provoquer le rejet d'un grand nombre de transactions lorsque l'ordre d'arrivée des opérations diffère fortement de l'ordre d'attribution des estampilles. De plus, les méthodes à base d'estampillage nécessitent une journalisation importante afin de pouvoir déterminer si les opérations arrivent ou non trop tard. Elles résolvent néanmoins les problèmes de famine par rajeunissement des estampilles.

8.3. Avantages et inconvénients du test de sérialisation

Nous pouvons émettre à propos du test du graphe de sérialisation à peu près les mêmes critiques que celles déjà formulées pour l'estampillage, à savoir une journalisation importante (en temps et en espace de stockage) pour le maintien des graphes de sérialisation, journalisation qui en outre doit remonter plus loin dans le temps que pour l'estampillage. Cette gestion alourdie permet cependant d'accepter plus d'opérations que l'estampillage et le verrouillage ne le permettaient.

---

[1] Notons qu'il n'existe pas de techniques de verrouillage exemptes d'interblocages à usage général pour les bases de données et qui soient très performantes. C'est ainsi que l'on trouve de multiples versions d'une même technique afin de l'adapter à tel ou tel cas particulier rencontré.

8.4. Avantages et inconvénients des méthodes optimistes

Les méthodes optimistes provoquent plus de redémarrages de transactions que les protocoles à verrouillage. Ceci est dû au fait que là où le verrouillage résout les conflits en bloquant la transactions conflictuelles, les méthodes optimistes résolvent les conflits en les avortant. De plus, dans le cas des méthodes optimistes, la non-sérialisabilité n'est détectée qu'au moment où les transactions demandent leurs terminaisons, gaspillant donc tout le temps passé à exécuter les opérations précédentes des transactions.

De plus, dans les méthodes à verrouillage, la recherche des interblocages se fait généralement à chaque demande d'opération conflictuelle. Les interblocages sont dès lors découverts plus tôt que dans les méthodes optimistes et le coût des avortements par conséquent souvent moins cher. C'est ce qui explique que les méthodes optimistes sont déconseillées pour des transactions relativement importantes (longues).

Les méthodes optimistes sont donc à envisager dans des environnements où les transactions sont courtes avec des probabilités de conflit faibles.

La recherche de techniques de verrouillage sans interblocages peut être vue comme une tentative pour réduire le coût du contrôle de concurrence en éliminant le défaire de transactions comme mécanisme de contrôle. L'optique optimiste est tout à fait à l'opposée de cela : elle cherche à éliminer le verrouillage. Ces méthodes optimistes comptent en effet sur le fait que les conflits seront rarissimes et par conséquent le nombre de transactions à défaire aussi. Toute la difficulté consistera à se rendre compte du conflit et à réparer les dégâts.

Puisqu'elles ne font pas usage du principe de verrouillage, elles ne provoqueront pas d'interblocages (le problème de la famine persiste néanmoins).

En outre, si les transactions sont en majorité lectrices, alors le surcoût de réparation devient presque négligeable.

8.5. Les schedulers intégrés

Nous avons vu trois grands mécanismes de synchronisation des opérations : le verrouillage, l'estampillage et le test du graphe de sérialisation (méthodes optimistes exceptées). Les points 8.1 à 8.4 ont permis d'éclaircir la situation quant aux avantages et inconvénients de chacun.

Jusqu'à présent, nous avons également traité la notion de conflit de façon tout à fait générale puisque nous y incluions aussi bien les conflits RW (ou WR) que les conflits WW. Il est cependant possible de décomposer le concept de sérialisabilité en distinguant ces deux types de conflit. On trouve dans [BERN81] des théorèmes en la matière, prouvant entre

autres que les conflits RW peuvent, sous certaines conditions, être résolus indépendamment des conflits WW.

Les deux remarques ci-dessus nous amènent tout naturellement à envisager l'utilisation d'une méthode particulière pour résoudre le premier sous-problème (synchronisation RW) et d'une autre pour le second sous-problème (synchronisation WW)[1], l'idée étant naturellement de profiter des avantages de chaque méthode sans devoir en subir les inconvénients.

Les modules intégrant deux méthodes différentes pour arriver à la double synchronisation (RW et WW) sont appelés **schedulers intégrés**. La grande difficulté liée aux schedulers intégrés sera de garantir l'intégration et la coopération correcte des deux sous-modules.

Avant de pouvoir enchaîner avec l'exposé plus détaillé de ces schedulers, nous devons redéfinir la notion de **conflit** puisque les conflits seront gérés au niveau de chaque sous-module indépendamment de l'autre.

Pour la synchronisation RW, deux opérations accédant à la même donnée sont en conflit si l'une est une lecture (R) et l'autre une écriture (W). Deux opérations d'écriture ne sont donc pas considérées comme conflictuelles à ce niveau !

Pour la synchronisation WW, deux opérations sont en conflit si elles pratiquent toutes deux des mises à jour.

*Nous utiliserons le terme "**synchroniseur**"[2] pour désigner l'un ou l'autre des deux sous-modules de synchronisation, RW ou WW.*

Pour les trois grands protocoles envisagés, à savoir 2PL, TO et SGT, voici l'esquisse des synchroniseurs correspondants :

◊ supposons qu'un synchroniseur S_1 soit implémenté selon un protocole **2PL** afin d'assurer la synchronisation RW. Lorsque S1 recevra d'une transaction Ti une demande d'écriture TR_WRITE_i (X, V), il ne devra la mettre en attente que si une autre transaction Tj détient déjà un verrou de lecture sur X. C'est-à-dire que de façon un peu paradoxale, plusieurs transactions pourront obtenir des verrous d'écriture sur un même objet tant que n'interviendra aucun verrou de lecture.

De façon similaire, si un synchroniseur S_2 est chargé d'assurer la synchronisation WW, il ne retardera jamais les demandes de lecture;

---

[1] Chacune de ces techniques est connue comme étant une technique de synchronisation. Celles qui résolvent les deux types de problèmes simultanément sont connues sous le nom de méthodes de contrôle de concurrence. Nous avions déjà essayé de faire apparaître cette nuance lors de l'introduction à ce travail.

[2] Traduction du mot anglais "synchronizer".

Cette distribution des tâches est correcte dans la mesure où chacun des deux synchroniseurs a été conçu en concordance avec l'autre.

◊ supposons maintenant un synchroniseur RW S_1 implémenté en **TO**. Il devra simplement garantir que deux demandes, l'une de lecture et l'autre d'écriture, soient traitées dans l'ordre des estampilles. Pour deux demandes d'écriture, S_1 ne devra pas imposer cet ordre, à moins que chacune d'elles n'entre en conflit avec une tierce opération de lecture.

Pour un synchroniseur WW S_2, le principe appliqué sera le même mais pour deux opérations d'écriture sur un objet commun;

◊ et enfin, pour des synchroniseurs de type **SGT**, chacun d'eux ne maintiendra un graphe de sérialisation que pour les types de conflit qu'il résout (soit RW soit WW). Le graphe de sérialisation RW d'un historique H aura pour noeuds les transactions confirmées dans H et un arc de T_i vers T_j si et seulement si, pour un objet commun,

- l'opération demandée par T_i est une lecture et précède une écriture de T_j, ou,
- T_i demande une écriture avant une lecture de la part de T_j.

Dans un graphe de sérialisation WW, il n'y aura un arc entre T_i et T_j que si toutes deux ont émis une opération d'écriture sur le même objet, celle de T_i précédant celle de T_j.

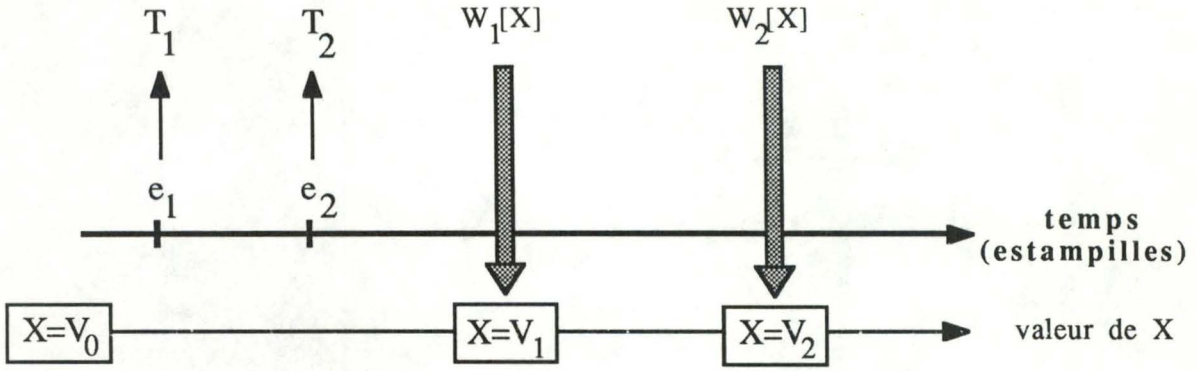
La grande difficulté sera de s'assurer que la séparation des deux graphes de synchronisation ne masque pas un cycle qui aurait existé s'ils étaient restés unis.

8.5.1. La règle d'écriture de Thomas : TWR

La **TWR** (Thomas Write Rule) est une norme en matière de règles de synchronisation WW. Elle est basée sur le protocole TO mais a pour objectif de ne jamais rejeter une demande d'écriture conflictuelle (avec une autre écriture).

En effet, en TO, étant données deux transactions T_1 et T_2 avec $e(T_1) < e(T_2)$, si un $W_2[X]$ a déjà été envoyé au DM et si le synchroniseur reçoit ensuite un $W_1[X]$, il doit défaire T_1.

La TWR s'oppose à cela en donnant l'argument suivant : si le $W_1[X]$ était malgré tout parvenu avant le $W_2[X]$ au synchroniseur, la valeur V_1 inscrite par T_1 dans X aurait de toutes façons été écrasée par la valeur V_2 de T_2 ainsi que l'indique la figure suivante :



De là l'idée que, même si la demande de T_1 arrive après que le synchroniseur ait envoyé celle de T_2 au scheduler, ce dernier ne la rejette pas mais procède comme si elle avait été reçue avant celle de T_2.

C'est-à-dire que lorsque le synchroniseur WW reçoit une demande d'écriture d'une transaction, si l'ordre des estampilles est respecté, il transmet la demande au DM, sinon, il fait comme si il l'avait envoyée au DM avant l'autre transaction et confirme l'exécution de la demande au TM (il n'envoie donc aucune opération au DM).

On peut se demander ce qu'il adviendra des lectures suite à l'application de cette règle.

Supposons en effet que nous ayons par ordre croissant d'estampilles, les opérations provenant de quatre transactions telles que :

$T_1 : W_1[X]$
$T_2 : W_2[X]$
$T_3 : R_3[X]$
$T_4 : W_4[X]$

et que le scheduler les reçoive dans l'ordre

$W_1[X]\ R_3[X]\ W_4[X] \parallel W_2[X]$

La TWR dit qu'après le ||, *le synchroniseur peut accepter la demande de T_2 sans toutefois la transmettre au DM; il doit simplement en confirmer la soi-disant exécution de T_2 au TM.*

Dans le déroulement normal des choses, cet accord n'aurait pas pu être donné. Le $W_2[X]$ aurait en effet dû être exécuté avant le $R_3[X]$, lui donnant ainsi à lire la valeur du $W_2[X]$ et non pas celle du $W_1[X]$.

Cette objection est bien sûr fondée, mais nous y répondons en argumentant que c'est là l'essence-même de la décomposition de la synchronisation en RW et WW.

Si nous construisions par exemple un scheduler dont le synchroniseur WW était du type TWR, ce dernier aurait plutôt tendance à déléguer les problèmes de synchronisation à l'autre synchroniseur, celui chargé des conflits RW.

Nous soulignons par là encore une fois l'importance de l'intégration des deux modules afin de fournir un scheduler complet et correct.

Les deux exemples de schedulers intégrés qui suivent sont proposés par Bernstein et al. [BERN87]. Le premier, avec synchronisation RW en TO de base et WW en TWR, est dit **pur** car les deux problèmes (RW et WW) sont réglés par le même mécanisme, l'estampillage. Le second est dit **mixte** car il combine la synchronisation RW en 2PL et la synchronisation WW en TWR.

8.5.2. Scheduler intégré pur

Le scheduler intégré que nous décrivons maintenant est composé d'un synchroniseur RW construit selon le principe du TO de base[1] et d'un synchroniseur WW utilisant la TWR.

Son comportement est le suivant :

- ◊ (TO) : accepter d'exécuter une lecture (écriture) si toutes les écritures (lectures) qui lui ont précédé proviennent de transactions plus âgées (jeunes); sinon, la rejeter;
- ◊ (TWR) : si l'opération demandée est une écriture et qu'il a déjà traité une écriture provenant d'une transaction plus jeune, en confirmer l'exécution sans toutefois la demander au DM; sinon, faire exécuter l'opération normalement par le DM.

Ce scheduler intégré est en quelque sorte une optimisation du scheduler TO de base dans laquelle une demande d'écriture tardive n'est rejetée que si une écriture conflictuelle plus jeune a déjà été traitée. A cette nuance près, leurs caractéristiques sont tout à fait comparables.

8.5.3. Scheduler intégré mixte

Nous avions évoqué lors de l'introduction consacrée aux schedulers intégrés la possibilité de construire un de type mixte, c'est-à-dire combinant deux techniques basées sur des concepts différents. Bernstein et al. [BERN87] en proposent un à base de 2PL pour la synchronisation RW et de TWR encore une fois pour la synchronisation WW.

---

[1] Rappelons qu'un scheduler TO de base traite une opération si toutes les opérations conflictuelles qu'il a traitées avant celle-là ont des estampilles plus petites.

Le fonctionnement de chacun des deux synchroniseurs ne nécessite pas d'éclaircissements particuliers. L'une et l'autre partie fonctionnant en effet indépendamment pour chaque tâche de synchronisation, leur protocoles sont simplement ceux présentés supra (voir présentation du 2PL et de la TWR).

Il faut néanmoins encore prouver que les deux synchroniseurs sont parfaitement intégrés. Il nous semble que l'apport de cette preuve, de par sa complexité, dépasse le cadre de notre travail. Si toutefois le lecteur est intéressé par cette explication, il peut consulter la référence [BERN87] que nous venons de citer.

CONCLUSION

Nous avons vu trois grandes approches pour résoudre les problèmes liés au parallélisme d'utilisation d'une BD : le verrouillage, l'estampillage et les méthodes optimistes. Quelques autres approches un peu moins conventionnelles sont aussi présentées, comme le test du graphe de sérialisation ou les schedulers hybrides dits "intégrés".

Le verrouillage assure la cohérence des exécutions en forçant les transactions à obtenir des verrous non-conflictuels sur des objets avant d'y accéder. Typiquement, deux requêtes sont en conflit si une des deux au moins demande une écriture, et une transaction sera bloquée si on lui refuse un verrou. Chaque fois qu'une transaction est mise en attente d'un verrou, elle risque d'entraîner un interblocage. Deux grandes optiques pour y remédier : la prévention et la détection. Le schéma préventif, traditionnellement à base d'estampillage (Wound-Wait ou Wait-Die), évite, par un système de priorités, que l'interblocage n'ait lieu. La détection quant à elle ne prend aucune mesure préalable mais compte sur une gestion de graphes, par exemple, pour détecter les interblocages et les réparer en tuant l'une ou l'autre transaction.

L'estampillage procède à un ordonnancement a priori des transactions et régule leurs accès en conséquence. De façon abrégée, chaque transaction reçoit une estampille pour toute sa durée de vie, estampille qui est donc renouvelée si la transaction est défaite et relancée. Classiquement, l'accès à un objet par une transaction ne sera acceptée que si aucune autre transaction plus jeune ne l'a déjà modifié.

Les schedulers intégrés tentent d'allier les avantages des différentes méthodes en faisant résoudre des sous-problèmes de synchronisation par chacune d'elles. Lors de la construction de ce type de scheduler, la difficulté est surtout de garantir que les parties travaillant bien individuellement collaborent aussi correctement.

Les méthodes optimistes autorisent les transactions à s'exécuter sans contraintes, mais pendant qu'elles s'exécutent, le SGBD rassemble des informations à leur propos. Lorsqu'elles demandent à être achevées, le système s'en sert pour procéder à la validation des opérations acceptées auparavant.

Le test du graphe de sérialisation gère en permanence un graphe des conflits de (preque) toutes les transactions présentes ou passées du système et cherche à éviter que des cycles n'y apparaissent.

Notre but n'était pas de donner Le "meilleur" algorithme de contrôle de concurrence. Celui-ci dépend en effet de l'environnement de la BD, du profil des transactions, selon que les transactions sont courtes ou longues, de simple consultation ou d'accès divers, etc...

Nous n'avons pas, à proprement parler, abordé les aspects "performance" des différentes méthodes. Ces considérations justifieront cependant une part importante des choix effectués par les utilisateurs d'un SGBD.

Alors que les algorithmes sont généralement bien compris, leurs performances et la prévision de leur comportement par contre, en particulier lorsque ces méthodes sont récentes, ne le sont pas toujours. Surgit alors la difficulté, lorsque nous touchons à des études de performances de SGBD, de comparer les résultats obtenus. Elle est notamment due au fait que les méthodes de contrôle de concurrence, que nous avons présentées dans le cadre de notre travail, sont basées sur des concepts très différents et mettent en oeuvre des moyens dont la charge n'est pas évaluable a priori. Le problème est encore aggravé lorsque les personnes menant les tests se basent sur des modèles qui leur sont propres et étudient des implémentations particulières de ces algorithmes.

Il serait dès lors souhaitable que l'on arrive à des standards, notamment au niveau des définitions et des tests. C'est pourquoi des travaux comme [FRAN85] méritent d'être répétés. Pour n'en reprendre que les points essentiels, Franaszek et Robinson définissent un niveau effectif E de concurrence, fonction du nombre n de transactions pouvant s'exécuter en parallèle sur un système et de la probabilité p d'apparition des conflits. Ils construisent un modèle de test de performances pour trois classes de méthodes de contrôle de concurrence : verrouillage, estampillage et méthodes optimistes.

Nous pouvons synthétiser leurs conclusions en disant que, pour une probabilité p donnée, lorsque n devient arbitrairement grand,

- ◊ E(verrouillage) tend à devenir nulle,
- ◊ E(estampillage) tend vers la constante 1/p,
- ◊ et E(méthodes optimistes) s'améliore constamment[1].

Ces résultats résument, selon nous, de façon concise l'ensemble des remarques que nous avons pu émettre à propos de ces trois méthodes.

Nous n'avons pas non plus considéré les questions d'implémentation des différents modules du SGBD. Pour une méthode à base de verrouillage, par exemple, étant donné le

[1] Considérant que les deux dernières techniques effectuent une partie importante de travail inutile pour des transactions qui sont finalement avortées, Franaszek et Robinson proposent trois alternatives basées sur des priorités et comparent leurs performances avec celles des trois méthodes classiques étudiées.

nombre d'accès dont il fait l'objet, il est crucial de fournir un gérant des verrous très performant. D'autres questions d'implémentation peuvent encore être soulevées, notamment celles à propos du niveau d'intégration du SGBD dans le système d'exploitation : quels avantages ou inconvénients y a-t-il à avoir un SGBD dissocié du système d'exploitation plutôt que bien intégré ?

PARTIE II

TECHNIQUES DE CONTROLE DE CONCURRENCE POUR BASES DE DONNEES DISTRIBUEES

Introduction

Nous avons soulevé les problèmes liés aux accès de plusieurs utilisateurs concurrents à une BD commune, via les transactions. Nous avons supposé que cette BD était gérée par un seul gérant de concurrence (un seul Transaction Manager et un seul Data Manager), c'est-à-dire que le système était supposé centralisé. Cette centralisation présentait le grand avantage qu'un seul responsable recevait toutes les requêtes des utilisateurs de la BD; il était donc parfaitement au courant de toutes les modifications que la BD pouvait subir. La gestion était à la charge d'un seul processeur et donc grandement simplifiée.

Il existe cependant de nombreuses situations dans la réalité où une BD n'est plus à la charge d'un seul processeur, mais bien de plusieurs processeurs communiquant entre eux. Il est tout à fait possible que n'importe lequel de ces processeurs (ou plutôt mécanismes de contrôle de concurrence tournant sur ces processeurs) ne gère pas la BD, la gère en totalité ou n'en gère qu'une partie. Ce dernier cas, naturellement le plus fréquent, est la raison d'être des SGBD distribuées.

Dans la suite de notre travail, nous présentons les algorithmes de contrôle de concurrence dans ce type de configuration, mais le lecteur peut présager que cette répartition des tâches n'est pas faite pour faciliter les choses. La gestion en sera d'autant plus difficile que de multiples utilisateurs peuvent accéder à des données stockées en divers endroits, qu'une même donnée pourra exister en plusieurs exemplaires et que le mécanisme de contrôle de concurrence s'exécutant sur un processeur ne peut instantanément avoir connaissance des interactions des autres processeurs.

Les problèmes de contrôle de concurrence pour les BD centralisées sont actuellement bien compris. Une théorie mathématique a d'ailleurs été développée pour les analyser et certaines solutions (comme par exemple le verrouillage à deux phases ou l'estampillage) ont été admises comme des normes. Il est de ce fait assez normal que les recherches en matière de BD distribuées se soient basées sur ces mêmes théories et méthodes. C'est ce que nous détaillerons dans la suite du travail. Toutefois, ces eaux sont encore assez troubles et les algorithmes proposés ne manquent pas. Nous essaierons d'y mettre un peu d'ordre, notamment grâce aux travaux [KOHL81], [BERN81], [BAYE82] et [BERN84].

Remarque :

Le contrôle de concurrence dans les systèmes centralisés est parfois qualifié de ***local****, et celui dans les systèmes distribués de* ***global****.*

Aux objectifs que nous avions attachés au contrôle de concurrence pour les BD centralisées[1], viennent s'en ajouter des nouveaux, notamment :

◊ la transparence de la localisation : il s'agit de fournir aux utilisateurs un service d'accès aux données sans qu'il ne leur soit demandé de connaître le processeur (plus tard nous parlerons plutôt de "site") responsable de la gestion de ces données;

◊ la transparence de la redondance : certaines données, pour des raisons de performances, devront parfois être stockées de façon redondante en plusieurs endroits du système, à nouveau sans que l'utilisateur n'en soit au courant.

Ces deux objectifs nécessiteront une gestion relativement complexe de la part du système de gestion de base de données distribuée, gestion des catalogues des emplacements physiques des données, notamment, ceci afin de permettre à la configuration de changer dynamiquement, par exemple, sans affecter la validité des applications. Nous supposerons que le système respecte déjà ces propriétés; il ne nous restera qu'à garantir que les algorithmes que nous proposerons les conservent bien.

[1] A savoir, principalement, la préservation de la cohérence des données et des temps de réponse finis (et raisonnables).

Chapitre 9

Le modèle de travail

Tout comme nous avions proposé un modèle de travail pour les SGBD centralisées, nous le faisons pour les SGBD distribuées. A lui seul il permet déjà de bien se rendre compte de la complexité du problème. Les éléments qui y sont utilisés (tels les DM et TM) ont les mêmes rôles que dans le modèle centralisé. Ce modèle est donné par la figure 9.1.

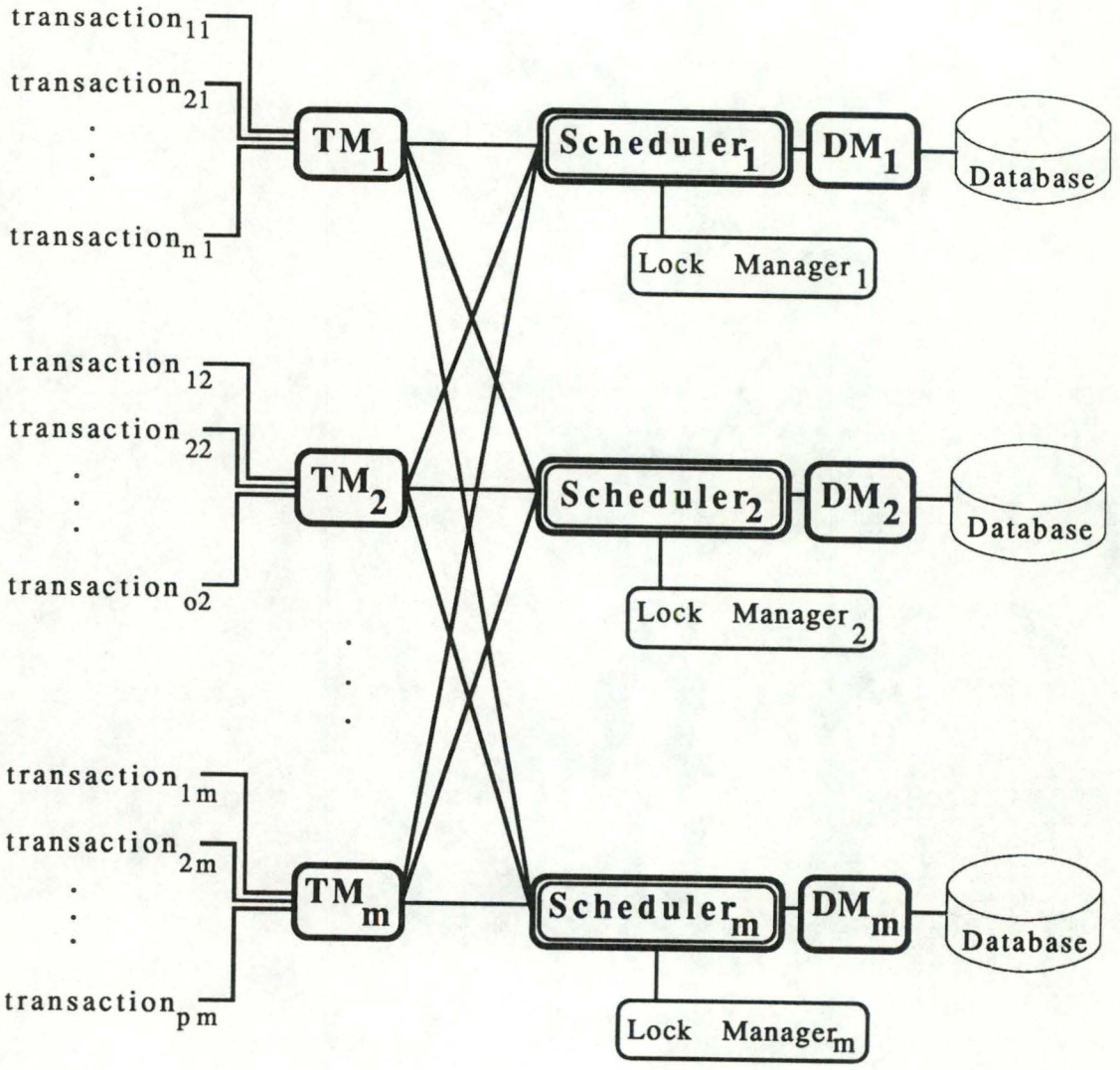


Figure 9.1 : Modèle de SGBD distribuée

Le **SGBDD** (SGBD Distribuée) est un ensemble de sites interconnectés par un réseau. Sur chacun de ces sites, s'exécutent un TM (Transaction Manager), un scheduler et un DM (Data Manager).

Le réseau est supposé parfaitement fiable : on garantit qu'un message envoyé par un site A vers un site B sera reçu par B et de façon non erronée. Ceci afin de ne pas encombrer notre exposé avec des problèmes qui ne concernent pas le contrôle de concurrence. Autre hypothèse simplificatrice : les messages sont reçus dans l'ordre de l'envoi.

Pour rappel, le **TM** reçoit les transactions des utilisateurs, une transaction n'étant gérée que par un et un seul TM pour toute sa durée de vie. Au niveau de cette interface, les quatre opérations que nous avions déjà répertoriées pour les BD centralisées sont encore valables : TR_BEGIN, TR_COMMIT, TR_READ (X) et TR_WRITE (X, nouvelle_valeur).

Après quelques opérations préparatives sur les demandes des transactions, le TM les transmet au **scheduler**, module responsable de tout le contrôle de concurrence. Chaque TM est donc supposé connaître l'emplacement d'une donnée (ou d'une de ses copies) parmi les sites[1].

Chaque scheduler utilisant le verrouillage, ainsi que c'était le cas pour le SGBD centralisé, communique avec un gérant des verrous, responsable de la pose et de la libération des verrous et capable de signaler au scheduler si le verrou demandé provoque un conflit ou non.

Lorsque le scheduler estime la chose faisable, il donne au **DM** l'ordre d'effectuer une opération physique de lecture/écriture. Ce dernier est donc responsable du stockage sur support stable de la BD.

Les TM sont capables d'échanger des messages avec n'importe quel DM du réseau. De même, un DM est capable d'en faire autant avec n'importe quel TM du réseau. Mais ni les TM ni les DM ne communiquent entre eux.

Ces échanges TM-DM devront toutefois être réduits au minimum puisque l'interconnexion des sites par un réseau risque d'augmenter fortement le coût des échanges de données.

[1] Il s'agit de ce qui est couramment appelé un "répertoire" (directory en anglais). Nous ne nous attarderons pas sur les manières qui s'offrent au TM pour y accéder.

Chapitre 10

La redondance des données

Dans la première partie de ce travail, nous avions consacré un chapitre aux données multiversions. La motivation d'une implémentation de contrôle de concurrence multiversion était d'augmenter le potentiel de parallélisme d'accès à un même objet en autorisant des transactions à accéder à des versions plus anciennes de cet objet si la versions récente était mobilisée par une transaction en mode conflictuel.

Dans les SGBDD, il est courant de trouver la duplication des données non pas en plusieurs versions dans le temps, mais en plusieurs versions dans l'espace. C'est-à-dire que la même valeur d'une donnée peut être stockée en plusieurs sites de façon à permettre aux transactions de ce site d'y accéder alors que d'autres transactions, en d'autres sites, se servent d'autres copies de cet objet. On parle dans ce cas de **redondance des données**.

Tout comme pour les données multiversions, les données redondantes sont destinées à augmenter la concurrence dans le système et celles sont donc supposées transparentes aux utilisateurs : le SGBDD doit masquer l'existence de cette multiplicité des copies d'une donnée.

Les principales motivations sous-jacentes à la volonté de multiplier les repésentations d'un même objet dans un système sont d'en augmenter la fiabilité et les performances d'accès.

La première est justifiée lorsque les données ont un caractère vital. La duplication de celles-ci permet en effet, en cas de destruction d'un support physique, de compter sur des copies de ces données pour pouvoir continuer une tâche importante.

La seconde vise à augmenter l'accessibilité des données. Dans une telle configuration, un utilisateur travaillant sur un site ne devra pas nécessairement faire ses accès via le réseau (accès lents), il pourra éventuellement disposer d'une copie locale de l'objet désiré.

L'utilisateur ne connaît que des objets logiques; chaque objet logique X peut

- ◊ soit être géré physiquement par un seul DM_k;
- ◊ soit avoir plusieurs représentations physiques X_i, chacune gérée par le DM_i correspondant.

Nous dirons que le scheduler$_k$ gère l'accès aux données stockées sur son propre site$_k$, et uniquement celles-là.

De manière intuitive, le lecteur réalise qu'il faudra pouvoir garantir que toutes les données physiques représentant une même donnée logique contiennent la même valeur.

Il suffit d'imaginer qu'un DM_1 fournisse à une transaction une valeur V_1 d'un objet X et qu'au même moment un DM_2 fournisse à une autre transaction une valeur V_2 alors qu'il s'agit du même objet X.

Les difficultés qui surgissent lors de l'introduction de cette multiplicité des représentations provoque un double problème :

◊ celui du contrôle de la concurrence

◊ et celui de la tolérance aux fautes.

Le problème du contrôle de la concurrence est celui qui nous concerne directement dans le cadre de ce travail. Nous passerons en revue les différentes techniques en la matière, en insistant principalement sur les difficultés introduites par la répartition et la duplication des données. Ces difficultés sont dues au souci de garantir ce qu'on appelle la **cohérence mutuelle** des copies d'un objet et qui signifie que "*toutes les copies d'un objet doivent converger vers la même valeur et devraient être identiques si l'activité de mise à jour cessait* " [KOHL81].

Le second problème, celui de la tolérance aux fautes, n'entre pas, a priori, en ligne de compte dans notre travail, étant donné le caractère spécifique au contrôle de concurrence que nous avons voulu lui donner. Quelques notions de tolérance aux fautes doivent cependant être introduites car elles influencent la conception des algorithmes de contrôle de concurrence.

De fait, si nous prenons par exemple une donnée qui se trouve stockée en n exemplaires sur n sites, la modification de l'un d'entre eux doit être répercutée sur tous les autres. Mais que se passera-t-il si à ce moment-là un des sites "est déjà en panne" ou "tombe en panne" ? Certaines copies risquent de ne pas être à jour.

Par abus de langage, nous considérerons le **scheduler distribué** comme l'ensemble des schedulers parmi tous les sites. Sa tâche sera de traiter les opérations requises par les transactions d'une manière globalement sérialisable (et recouvrable, mais nous n'abordons pas ce point dans ce travail).

Lorsque l'exécution parallèle de plusieurs transactions dans une BD à exemplaires multiples sera équivalente à une exécution sériale de ces transactions dans une BD à

exemplaire unique, nous dirons que cette exécution est ***1-sérialisable*** *("One-copy serializable" en anglais).*

Les problèmes soulevés par la redondance des données peuvent être abordés de deux manières : la première, ne tenant pas compte de l'éventualité d'une panne d'un site, la seconde en tenant compte. Ce sont respectivement les approches "Write-All" et "Write-All-Available" présentées ci-dessous.

10.1. L'approche Write-All

Deux grandes approches permettent d'aborder le problème de la redondance des données, dont la première est appelée "approche Write-All" et que cette section s'évertue à présenter.

Supposons qu'une donnée X soit stockée en m sites distincts et que ces copies soient, par convention, notées X_1, X_2 ... X_m.

Alors,

◊ pour un ordre **TR_READ** (X), il suffira que cette demande soit transmise à UN SEUL des schedulers disposant d'une copie de X, soit X_k avec ($1 < k < m$); ce scheduler fonctionnera alors comme un scheduler local unique;

◊ pour un ordre **TR_WRITE** (X, V), il faudra que cette demande soit transmise à TOUS[1] les schedulers disposant d'une copie de X.

C'est pourquoi cette approche peut être qualifiée de Write-All.

Malheureusement, ainsi que nous l'avions signalé lors de l'introduction aux problèmes posés par la redondance, certains sites peuvent être en panne au moment de cette phase d'écriture généralisée. Il est alors possible que les mises à jour ne soient pas reportées sur les sites en panne; lors de leur "réveil", ils fourniront par conséquent aux transactions des valeurs périmées.

Si nous voulons absolument nous en tenir à l'approche Write-All, nous devons accepter la solution radicale (et presque inadmissible) de retarder les TR_WRITE jusqu'à ce que l'écriture généralisée puisse s'exécuter sur TOUS les sites concernés. La littérature parle alors de "propagation des mises à jour".

---

[1] C'est pourquoi l'approche est qualifiée de "Write-All".

Une conséquence immédiate est qu'au plus nombreuses sont les copies d'un objet, au plus forte est la probabilité qu'une d'entre elles soit indisponible. La redondance va dans ce cas directement à l'encontre de son objectif de meilleure disponibilité.

Nous nous devons donc d'envisager la solution plus satisfaisante qui suit.

10.2. L'approche Write-All-Available

Le problème de l'approche Write-All étant finalement de vouloir mettre à jour toutes les copies d'une donnée, il nous semble normal d'envisager l'approche **Write-All-Available,** qui n'actualise que les copies disponibles. L'objectif de disponibilité reste donc satisfait aux dépens parfois de celui de l'exactitude des données fournies.

En effet, prenons l'exemple suivant :

- *soit une donnée X dont 2 copies X_1 et X_2 sont stockées sur les sites S_1 et S_2,*
- *et une transaction T_1 qui vient d'effectuer une mise à jour de X répercutée correctement sur les deux sites;*
- *supposons que le site S_1 tombe en panne et qu'une nouvelle transaction T_2 tente de modifier X; par le principe du Write-All-Available, seule sera modifiée la copie X_2.*

Conclusion :

Après restauration du site S_1, la prochaine transaction qui obtiendra la copie X_1 en réponse à une demande de lecture de X, n'obtiendra pas la valeur correcte de X (qui devrait être celle de la dernière transaction exécutée, T_2) mais une valeur périmée ne provenant pas de la dernière transaction qui a effectué une modification de X. Cette exécution n'était donc pas 1-sérialisable, selon la définition que nous en avons donnée supra.

Il existe plusieurs variantes de l'approche Write-All-Available et autres méthodes destinées à éviter de telles exécutions incorrectes.

Convention :

Lors de la présentation des différentes méthodes qui vont suivre, nous comptons sur le titre des sections et sous-sections pour permettre au lecteur de distinguer si l'algorithme présenté traite ou non la redondance des données, ceci afin d'alléger la lecture de notre travail et d'offrir des points de comparaison entre les algorithmes avec ou sans redondance.

Chapitre 11

Le verrouillage à deux phases : D2PL

Une technique importante, si pas la plus importante, de contrôle de concurrence dans les SGBD centralisées verrouillage, à savoir le verrouillage, est encore d'application lorque nous travaillons en milieu réparti.

Le principe du verrouillage centralisé se résume comme suit :

◊ une transaction peut verrouiller un objet pour en assurer l'inaccessibilité pendant la mise à jour (qui le met temporairement dans un état incohérent);

◊ les verrous sont le plus souvent de deux types : exclusif ou partagé. Deux verrous seront en conflit lorsqu'un des deux au moins est exclusif. Lorsque la demande de verrouillage d'une transaction provoque un conflit, celle-ci doit soit abandonner, soit attendre, soit s'imposer;

◊ une transaction qui avorte en cours d'exécution (à cause d'un ABORT) doit restaurer la BD dans l'état qui lui précédait; afin d'éviter le problème de la cascade des avortements ou de l'incohérence statistique, une transaction doit par conséquent conserver ses verrous exclusifs jusqu'au TR_END;

◊ un interblocage peut surgir lorsque deux transactions ou plus s'attendent mutuellement. Les méthodes de prévention permettent de les éviter, les méthodes de détection de les réparer.

Parmi toutes les techniques qui respectent ces conventions, celle du verrouillage à deux phases est la plus répandue. Le 2PL centralisé (cfr. supra la section 3.2 consacrée à ce sujet) garantit en toutes circonstances la préservation de la cohérence à condition que les transactions respectent le protocole 2PL, à savoir qu'elles soient constituées d'une phase de croissance (acquisition des verrous) suivie d'une phase de récession (libération des verrous). Dans la suite de notre travail, nous utiliserons l'abréviation **D2PL** (Distributed 2PL) pour désigner la version distribuée du verrouillage à deux phases.

Ce rappel étant fait, nous pouvons maintenant en venir au coeur de cette section qui est de présenter en quoi le 2PL local peut servir de base au D2PL (ou 2PL global) sans toutefois résoudre tous les problèmes de la répartition des données. Nous verrons alors quelles modifications doivent lui être apportées.

11.1. Version D2PL sans redondance

Puisque nous avons spécifié les propriétés des historiques générés par le scheduler indépendamment du caractère distribué ou non du SGBD (transparence de localisation), les preuves valables pour le 2PL centralisé le restent pour le D2PL.

Lorsque le scheduler reçoit les opérations de lecture/écriture sur un objet X d'une transaction, il les transmet (au moment propice) à son DM local qui a directement accès à la valeur stockée de X. Chaque scheduler est par conséquent capable de gérer les verrous pour les données stockées sur son site.

Lorsqu'il reçoit un TR_END, il l'envoie à tous les sites sur lesquels la transaction a effectué des accès.

En 2PL, une opération sur un objet ne peut être effectuée que si le verrou adéquat a été obtenu, ce qui dépend uniquement des autres verrous déjà posés sur cet objet. Chaque scheduler local, travaillant comme un scheduler 2PL centralisé unique, dispose donc des renseignements nécessaires pour décider de faire ou non exécuter une opération, sans qu'il ait à communiquer avec d'autres sites.

La libération des verrous, contrairement à leur acquisition, est plus problématique. Plus précisément, le moment où cette libération intervient pose problème.

En effet, de par le protocole 2PL, un scheduler ne peut libérer les verrous d'une transaction que lorsqu'il sait que celle-ci ne soumettra plus aucune opération à lui ou à n'importe quel autre scheduler. Imaginons en effet l'exemple suivant :

un scheduler S_1 libère un verrou V_1 pour une transaction T_i, et un peu plus tard un autre scheduler, S_2, pose un verrou V_2 pour la même T_i, enfreignant de ce fait la règle des deux phases.

Il semblerait donc que cette exigence requière une communication entre sites (afin de savoir si une transaction donnée y soumet encore des opérations).

C'est vrai à moins qu'on n'utilise la **version stricte du D2PL**. En effet, au moment où le scheduler décide d'envoyer le TR_END_i d'une transaction T_i à tous les sites dans lesquels elle a accédé à des données, il doit avoir reçu confirmation de l'exécution de toutes les opérations de T_i. Ceci présuppose donc que la transaction avait obtenu tous les verrous nécessaires. Et par conséquent, si le scheduler libère les verrous immédiatement après avoir envoyé les demandes de COMMIT[1] aux sites concernés (sans attendre une quelconque confirmation), il est certain qu'aucun scheduler ne recevra plus d'opérations de T_i.

[1] Conformément au protocole 2PL strict.

11.2. Versions du D2PL avec redondance

Il est normal d'envisager que la redondance dans le stockage d'une information complique la gestion des données, surtout si l'on veut assurer la transparence de la redondance vis-à-vis des utilisateurs.

Les protocoles que nous présentons maintenant sont tous à base de D2PL et résolvent les problèmes de redondance. Toutefois, ces implémentations suivent l'approche Write-All ce qui signifie que pour être correctes, elles doivent attendre que tous les sites soient disponibles en cas de mises à jour.

11.2.1. Version de base du D2PL

Le protocole rudimentaire basé sur le 2PL consiste simplement à considérer que pour chaque copie physique d'un objet, le scheduler attaché au DM qui le gère est responsable de l'accès à cet objet. Chaque scheduler local fonctionnera alors comme un seul scheduler tout à fait indépendant des autres.

Nous devons encore nous assurer que l'historique produit par l'ensemble des schedulers respecte bien le protocole 2PL (phase de croissance et phase de récession) afin d'arriver à une 1-sérialisabilité des exécutions.

En procédant de manière Write-All, étant donné la transparence à la duplication des données de notre approche, il suffit de respecter le protocole D2PL défini à la section 11.1 pour arriver à une version distribuée du 2PL traitant correctement la redondance.

11.2.2. D2PL Centralisé

Il peut paraître absurde de parler d'une version centralisée du D2PL alors que nous nous intéressons aux protocoles distribués. Cet abus de langage signifie qu'au lieu d'avoir un scheduler indépendant sur chaque site, on a préféré les regrouper en un seul scheduler centralisé à qui incombe toute la responsabilité du contrôle de concurrence.

Cette solution est envisageable dans la mesure où elle permet de réduire le nombre de messages à échanger lors de l'accès aux données (demande de verrouillage, accord de verrouillage, demande de mise à jour, accord de mise à jour, libération de verrous, etc...) et évite les interblocages globaux.

Elle présente cependant deux inconvénients majeurs :

◊ le scheduler central risque de devenir un goulet d'étranglement,

◊ et, en cas de panne de ce scheduler, c'est tout le système qui s'effondre; bref, exactement le contraire de ce qu'on attendait dans les SGBDD !

C'est pourquoi il vaut sans doute mieux ne pas envisager cette solution et s'orienter vers d'autres possibilités.

11.2.3. D2PL avec Copies Principales

Le **principe de la copie principale** consiste à choisir, parmi toutes les copies physiques d'une donnée, un représentant appelé précisément "copie principale". Avant de pouvoir accéder à une quelconque copie X_i d'une donnée X, la transaction doit obtenir un verrou sur sa copie principale X_p.

Pour la lecture, cette condition impose donc plus d'échanges (via le réseau) que le D2PL de base.

En effet, supposons qu'une transaction T veuille accéder à une copie X_i autre que X_p (peu en importe la raison). T devra donc communiquer avec deux DM : celui qui gère X_p, pour pouvoir le verrouiller, et celui qui gère X_i, pour pouvoir y accéder; en D2PL de base, T n'aurait dû communiquer qu'avec ce dernier.

Pour l'écriture par contre, cette exigence du verrouillage de la copie principale n'exige pas plus de communications que la version de base.

En D2PL de base, une transaction voulant devenir rédactrice sur une donnée X devait obtenir un verrou exclusif sur chaque copie X_i avant de modifier chacune d'elles.

Par le procédé des copies principales, la même transaction doit simplement obtenir le verrouillage exclusif de X_p, copie principale de X, avant de modifier bien sûr chacune des copies X_i.

L'apport de cette variante du D2PL n'est pas vraiment évident : le peu qu'elle gagne en écriture, elle le perd en lecture. De plus, contrairement à l'approche centralisée (cfr. sous-section 11.2.2), elle n'exclut pas les interblocages. Enfin, lors de la panne d'un site responsable de la copie principale d'une donnée X, alors X devient indisponible, du moins pour une mise à jour alors qu'il peut encore exister d'autres copies disponibles de X.

On pourrait toutefois lui reconnaître le mérite de "cacher" quelque peu la multiplicité des exemplaires par le principe de la copie principale et donc d'aider à l'objectif de transparence de la redondance.

11.2.4. Algorithme des copies disponibles : ACA

Les algorithmes dits "des copies disponibles" **ACA** (Available Copies Algorithms) utilisent une forme améliorée de l'approche Write-All-Available, c'est-à-dire que chaque

lecture peut se faire sur n'importe quelle copie d'une donnée alors que l'écriture doit se faire sur toutes les copies de cette donnée.

Rappelons que l'approche Write-All-Available à elle seule ne garantissait pas la 1-sérialisabilité.

Cette méthode suppose que chaque site est capable de savoir si un autre est en panne, simplement en lui envoyant un message (mais pas directement). Le scheduler utilisé est de type D2PL strict, à savoir qu'après l'accès d'une transaction T à une copie d'une donnée, aucune autre transaction ne peut encore accéder à cette donnée dans un mode conflictuel jusqu'à ce que T se soit terminée.

De plus, l'ensemble des copies d'une donnée est supposé ne pas changer dynamiquement et être connu de chaque site. Enfin, chaque donnée ne doit être créée et ne peut tomber en panne qu'une seule fois.

Conventions :

Nous désignerons les sites par les lettres A, B, C etc.. et X_A pour reconnaître la copie d'une donnée X stockée en un site A. Lorsqu'une transaction T demandera la lecture d'une donnée X, le TM du site émetteur de T choisira une des copies de X, soit X_A, et enverra la demande de lecture $R[X_A]$ au scheduler; le TM choisira généralement d'effectuer l'accès à la copie sur le site le plus proche, le sien de préférence, ceci afin de réduire les coûts de communication.

Nous qualifierons une copie X_A d'"initialisée" lorsqu'une demande d'écriture a été traitée, même si la transaction rédactrice n'a pas encore été confirmée (par un COMMIT).

Nous pouvons, suite à ces conventions, donner les règles constitutives du **protocole ACA** que voici :

◊ si une transaction T_i demande une lecture de X, le TM du site émetteur de T_i enverra donc l'ordre de lecture $R[X_A]$ au site A, et

- si le site A est opérationnel et X_A est initialisée, le $R[X_A]$ sera traité par les scheduler et DM de ce site, et,

  ° si X_A a été initialisée, par la transaction T_j par exemple, mais que T_j n'a pas encore été confirmée, alors, par la règle du D2PL strict, le $scheduler_A$ doit retarder cette lecture jusqu'à ce que T_j se termine (par un TR_COMMIT ou TR_ABORT);

  ° si $R[X_A]$ doit être rejetée à cause d'un conflit, il n'est inutile d'essayer de satisfaire T_i avec une autre copie de X puisqu'il y aurait également conflit; T_i doit alors être avortée;

° si $R[X_A]$ est acceptée, la valeur lue sera renvoyée au TM de T_i et la lecture de X considérée comme fructueuse;

- si par contre le site A est en panne ou la copie X_A n'a pas été initialisée, alors, après un certain délai d'attente, le TM s'en rendra compte et pourra, soit avorter T_i, soit, et c'est préférable, tenter de soumettre la lecture de X à un autre site : si un tel site n'existe plus, alors T_i doit quand-même être avortée, sinon, la lecture de X sera réussie;

◊ si T_i émet une demande d'écriture, le TM envoie des ordres d'écriture $W[X_Z]$ à tous les sites Z qui sont censés détenir une copie de X, et,

- si le site Z est en panne, le TM, après un certain délai, finira par s'en rendre compte et renverra un signal de refus à T_i;

- si le site Z est opérationnel,

° si X_Z a été initialisée, le $W[X_Z]$ sera traité par les scheduler et DM du site Z et la réponse correspondante envoyée au TM de T_i;

° si X_Z n'a pas été initialisée, le $W[X_Z]$ provoque l'initialisation de X_Z et la réponse correspondante renvoyée au TM de T_i.

- enfin, après avoir envoyé tous les ordres de lecture aux sites détenant des copies de X, le TM attend leurs réponses;

° si tous les ordres envoyés sont acquiescés, alors la lecture est considérée comme réussie;

° si, par contre, un des sites a émis un refus ou si tous sont restés muets[1],alors la lecture demandée par T_i doit être avortée.

Notre protocole ACA est maintenant presque complet, il ne nous reste plus qu'à en garantir la 1-sérialisabilité, puisque nous avions fait remarquer à propos de l'approche Write-All-Available disant qu'elle n'était pas 1-sérialisable.

Le protocole ACA doit donc être complété par un protocole de validation. Cette validation commence après que les réponses aux demandes de la transaction émettrice T_i aient été reçues. A ce moment, T_i connaît toutes les demandes d'écriture qui n'ont pas reçu de réponse ainsi que toutes les copies auxquelles elle a vaiment accédé.

Cette validation se fait en deux pas :

[1] C'est-à-dire n'ont pas envoyé de réponse, et donc peuvent être considérés comme en panne.

◊ T_i s'assure que toutes les copies auxquelles elle a accédé mais pour lesquelles elle n'a pas reçu de réponses sont encore disponibles, et,

◊ T_i s'assure que toutes les copies pour lesquelles elle a réussi un accès sont encore disponibles.

Cette validation se fait par envoi de messages aux différents sites. Ceci suggère que le protocole ACA requiert un échange de messages important. Les techniques permettant de réduire ce dernier sont donc utilisées au maximum, dans la mesure du possible. Pour des renseignements complémentaires à propos de ce protocole ACA, la référence en matière de preuve d'algorithmes de contrôle de concurrence, [BERN87], peut à nouveau être conseillée au lecteur.

11.3. Les interblocages dans les SGBDD

Il est bien évident que le problème des interblocages présent dans les BD centralisées ne fait que se complexifier dans une configuration répartie.

Si nous considérons chaque scheduler comme autonome, les interblocages locaux sont résolus par les méthodes vues pour les BD centralisées.

Mais persiste encore la possibilité d'interblocages globaux, c'est-à-dire impliquant plusieurs sites.

Reprenons l'outil fondamental en matière d'interblocages, le graphe des mises en attente[1] (WFG), et admettons que chaque scheduler maintienne un WFG local afin de pouvoir détecter et résoudre les interblocages locaux. Si nous appelons **WFG global** l'union de tous les WFG_i locaux, nous pouvons montrer à l'aide d'un exemple qu'il peut exister des cycles au niveau du WFG global alors qu'il n'y en aucun au niveau des WFG_i locaux.

Soit les deux transactions définies comme suit :

T_1 :	TR_BEGIN_1	T_2 : TR_BEGIN_2
	$TR\_READ_1\ (X)$	$TR\_READ_2\ (Y)$
	$TR\_WRITE_1\ (Y, V_1)$	$TR\_WRITE_2\ (X, V_2)$
	TR_COMMIT_1	TR_COMMIT_2

et supposons que T_1 soit prise en charge par le scheduler S_1 et T_2 par le scheduler S_2; nous pouvons imaginer le scénario (ou historique) suivant :

- S_1 pose $RL_1[X]$ (pour pouvoir plus tard satisfaire le TR_READ_1);

[1] Cfr. la sous-section 3.5.2 consacrée aux interblocages, dans la première partie de notre travail.

- *S_2 pose $RL_2[Y]$ (pour pouvoir ensuite satisfaire le TR_READ_2);*

- *S_1 devrait poser $WL_1[Y]$ pour le TR_WRITE_1 mais doit mettre T_1 en attente car ce verrou entre en conflit avec le $RL_2[Y]$ de T_2. S_1 ajoute donc un arc de T_1 vers T_2 dans son WFG_1 local;*

- *de même, S_2 devrait poser $WL_2[X]$ pour T_2 mais doit la mettre en attente car il existe déjà un verrou $RL_1[X]$. S_2 ajoute donc un arc de T_2 vers T_1 dans son WFG_2 local.*

Nous nous trouvons donc dans une situation où aucun des schedulers locaux n'a détecté d'interblocage dans son propre WFG, alors qu'il y a un cycle dans le WFG global.

Il est par conséquent nécessaire d'arriver à une solution qui détecterait ou préviendrait ce type d'interblocage appelé **interblocage distribué**.

11.3.1. Détection des interblocages

Rappelons simplement que la détection des interblocages consiste à laisser les transactions s'attendre l'une l'autre sans aucun contrôle et de n'en avorter que si un interblocage s'est réellement produit. Pour de plus amples explications quant aux particularités des méthodes de détection, nous renvoyons le lecteur à la section correspondante dans la partie consacrée aux BD centralisées.

Nous pouvons dès lors passer directement aux techniques en question.

11.3.1.1. Approche centralisée

Tout comme nous avions un peu paradoxalement proposé une version centralisée du D2PL, nous proposons un détecteur d'interblocages centralisé.

Nous venons de faire remarquer que l'interblocage n'était décelable que dans le WFG global et pas dans les graphes locaux. De là l'idée d'attribuer à une autorité indépendante la gestion de ce graphe global. Ce module recevrait "périodiquement" les morceaux de WFG de l'ensemble des schedulers locaux et reconstituerait à partir de là le WFG pour l'ensemble du SGBDD.

Nous utilisons l'adverbe "périodiquement" car il nous semble excessif (et coûteux) de maintenir cet arbre global en permanence à jour, notamment à cause des délais relativement plus longs pour obtenir les informations demandées aux différents sites (ces informations passent en effet par le réseau).

Dès lors, les critiques seront les mêmes que celles formulées à propos d'une détection tardive d'un deadlock dans les SGBD centralisées, à savoir surtout que les transactions en

situation de deadlock bloqueront leurs données déjà verrouillées jusqu'à ce que cette situation soit détectée, alors qu'elles ne les utilisent plus.

Une fois l'interblocage détecté, le schéma est semblable à celui suivi pour les BD centralisées : choisir une victime, la défaire et éventuellement la refaire.

Le problème du choix de la victime est à nouveau délicat car le détecteur d'interblocages central n'a a priori pas d'informations quant aux candidats-victimes. Le coût de l'échange de messages entre le détecteur central et les différents schedulers étant fortement conditionné par le nombre de messages échangés, les schedulers locaux auront souvent pour habitude de communiquer, en même temps que leur WFG, un candidat-victime au cas où il y aurait interblocage. Chaque scheduler est donc capable d'évaluer les coûts relatifs au défaire des transactions locales sur la base des critères que nous avons définis pour le cas des interblocages centralisés.

Remarque :

Nous avions fait remarquer que les interblocages dans les SGBD centralisées n'impliquaient généralement que peu de transactions. Il est possible de prouver que cette affirmation est encore renforcée dans le cadre des SGBDD.

Cette considération nous amène à la réflexion suivante : puisque le coût des échanges de messages est assez élevé, le détecteur global ne collecte les informations en provenance des autres qu'à intervalles réguliers. Un interblocage distribué peut donc ne pas être détecté pendant un certain temps. Ceci est particulièrement gênant lorsque les interblocages ne concernent que très peu de transactions puisqu'on met en oeuvre des moyens importants pour ne résoudre que des problèmes limités.

*De là l'idée de faire communiquer les sites entre eux qui pourraient ainsi détecter ce type d'interblocages plus rapidement. Bernstein et al. détaillent une méthode de détection des interblocages distribués, baptisée "**Path Pushing**", qui permet d'alléger cet échange d'informations intersites (car tous les sites peuvent communiquer avec tous les autres). Nous ne la soumettons pas au lecteur, notamment parce que notre modèle n'admettait pas les échanges directs entre sites.*

Les inconvénient majeurs de la centralisation sont bien sûr la vulnérabilité des autres sites, puisqu'ils dépendent tous du bon fonctionnement du site responsable de la détection des interblocages, et la perte d'autonomie de ces sites.

11.3.1.2. Approche hiérarchique

Dans une approche hiérarchique, les sites du système sont organisés en hiérarchie avec un détecteur d'interblocages à chaque noeud de l'arbre. Les interblocages locaux à un seul site sont alors détectés par ce site-même, les interblocages impliquant deux ou plusieurs

cycles sont détectés par le noeud (site) ascendant dans l'arbre. Badal [BADA86] propose un exemple d'algorithme de détection des interblocages à structure hiérarchique et en étudie les performances.

A cette différence près, les reproches que l'on peut faire à cette approche sont tout à fait semblables à ceux formulés pour l'approche centralisée. Les approches nécessitent d'ailleurs toutes deux des échanges de messages entre un ou plusieurs sites de détection.

A titre documentaire, citons encore deux autres articles [OBER82] et [SINH85] qui présentent également des algorithmes de détection d'interblocages. Le premier fonctionne selon le principe de l'analyse des graphes de mises en attente, mais pas le second, qui procède par attribution de priorités.

11.3.2. Prévention des interblocages

Rappelons que la prévention des interblocages est une approche prudente dans laquelle le SGBDD redémarre un transaction s'il craint qu'un interblocage puisse se produire.

11.3.2.1. Implémentation en D2PL

Afin d'implémenter la prévention des interblocages sur des schedulers D2PL, nous devons les modifier quelque peu.

Supposons que T_i soit la transaction émettant la requête et T_j, celle qui détient déjà un verrou sur le même objet, et admettons encore que le scheduler soit capable, par exemple sur base d'une analyse de graphes de mises en attente, de mener un test afin de déterminer si l'obtention d'un verrou risque de provoquer un interblocage. La modification apportée au scheduler est alors la suivante : lorsqu'une demande de verrouillage de T_i est refusée, le scheduler vérifie, par le test, si T_i et T_j sont admissibles, et,

- ◊ si T_i et T_j réussissent le test, alors T_i est mise en attente de T_j comme d'habitude;
- ◊ sinon, une des deux transactions est redémarrée : si c'est T_i, l'algorithme est dit "sans préemption"; si c'est T_j, "avec-préemption".

Une manière normale de mener le test serait de vérifier que l'acceptation de la mise en attente de T_i ne provoquera pas d'interblocage en recherchant la présence de cycles dans le graphe des mises en attente. Nous avons déjà signalé (à la sous-section 11.3.1.) quelle difficulté la répartition des sites amenait.

Une autre façon d'empêcher l'occurence d'interblocages serait de purement et simplement ne jamais accepter la mise en attente d'une transaction.

Rien de surprenant à ce que, suite à ces remarques, nous proposions des méthodes préventives plus pratiques, à base de priorités, ainsi que nous l'avions fait dans la première partie.

11.3.2.2. Les méthodes prioritaires

Nous passerons brièvement en revue le principe des priorités comme solution préventive aux interblocages.

Il s'agit d'accorder des priorités aux transactions et de tester ces priorités afin de décider si une transaction T_i peut en attendre une autre, T_j. On peut par exemple accepter que T_i attende T_j si sa priorité lui est inférieure, sinon, refuser cette mise en attente (ou inversément). Ce test empêche la formation d'interblocages car classe les transactions selon un ordre strict, et il serait par conséquent impossible qu'une transaction se trouve impliquée dans un cycle puisque cela signifierait qu'elle a une priorité qui lui est strictement supérieure.

L'inconvénient de cette méthode est, rappelons-le, de ne pas préserver le système des situations de famine, dans lesquelles une transaction se voit systématiquement défaite et attribuer une nouvelle priorité. C'est pourquoi des priorités plus particulières sont introduites, les estampilles. Le point suivant leur est consacré.

11.3.2.3. L'estampillage et les problèmes de synchronisation

Pour résumer le principe de l'estampillage vu dans la première partie, nous dirons que :

- ◊ l'estampille est un numéro attribué une fois pour toutes[1] à chaque transaction;
- ◊ avant d'autoriser une transaction T_i d'attendre une autre transaction T_j, le scheduler compare leurs estampilles. Si T_i est plus prioritaire que T_j, on lui accorde l'attente et on ajoute un arc de T_i vers T_j dans le WFG, sinon on l'avorte;
- ◊ puisque les transactions sont classées selon un ordre strict, il est impossible d'arriver à un cycle dans le WFG et donc à un interblocage.

Lorsqu'il n'y a qu'un seul TM dans le système, il lui est facile de générer des estampilles uniques et par ordre croissant. Toutefois, lorsque comme dans un système distribué plusieurs TM cohabitent, il devient nécessaire d'arriver à une **synchronisation** entre les TM de façon à ce que l'ensemble des estampilles respecte les critères d'unicité et de stricte monotonie. Nous devons donc trouver une méthode qui satisfasse à cette exigence et qui

[1] Dans le cadre des techniques de prévention des interblocages ! Car l'estampillage en tant qu'alternative du verrouillage peut parfois renouveler l'estampille d'une transaction.

n'oblige pas les TM à communiquer entre eux (nous avons écarté cette hypothèse dans notre modèle et cela exigerait de toutes façons des communications trop coûteuses pour la génération des estampilles).

Une technique simple consiste à donner à chaque TM un identifiant, son numéro de site par exemple. Il n'assignerait plus à chaque transaction uniquement l'estampille locale, mais la paire consituée du numéro de TM et de l'estampille locale qu'il génère. La première partie de cette estampille globale étant par définition unique, il suffit d'assurer que la seconde le soit aussi. On propose souvent l'horloge comme générateur automatique d'estampilles locales; il suffit alors de s'assurer qu'il y a bien eu un "tic horloge" entre deux prélèvements pour des estampilles. Plus facile et sans doute plus rapide, le TM peut aussi utiliser un simple compteur qu'il incrémente lui-même à chaque nouvelle estampille.

Unicité et stricte croissance des estampilles étant maintenant acquis, chaque scheduler local peut travailler tout à fait indépendamment des autres et appliquer le principe de l'estampillage tel que présenté dans le cadre des BD centralisées.

11.3.2.4. Des alternatives

L'estampillage est certainement la méthode préventive aux interblocages la plus courante. Quelques alternatives à l'estampillage ont cependant été présentées dans la première partie.

Nous avions par exemple proposé le préordonnancement des ressources ou la prédéclaration des données accédées en lecture et écriture. Ces méthodes ont pour principaux inconvénient d'imposer une prédéclaration des listes d'intention, ce qui n'est pas possible dans de nombreuses applications, ou de forcer l'acquisition ds verrous dans un ordre séquentiel.

Nous n'insisterons pas sur ces méthodes qui nous semblent peu attrayantes.

Chapitre 12

L'estampillage : DTO

Nous présentons dans ce chapitre une des méthodes de contrôle de concurrence les plus importantes et qui ne fonctionne pas selon un principe de verrouillage. Il s'agit de l'estampillage. Seule des versions traitant des données à copie unique seront présentées. Les problèmes dus à la multiplicité des copies ont été convenablement développés dans les chapitres ci-dessus et l'hypothèse de transparence de la duplication des données nous permet de ne plus y revenir dans les chapitres suivants.

Nous ne reviendrons pas non plus sur les règles gouvernant les protocoles d'estampillage; le chapitre 4 de la première partie et le point 11.3.2.1 de cette seconde partie permettront au lecteur de se les remémorer, si nécessaire.

12.1. Protocole DTO de base

Les schedulers TO (de deux politiques, Wait-Die et Wound-Wait) exposés dans la première partie furent en fait spécialement conçus pour des systèmes décentralisés. C'est ce qui explique la faculté d'installation de ces algorithmes sur ces systèmes.

Chaque site dispose de son scheduler de type DTO, de façon similaire au D2PL; chacun gère l'accès aux données stockées sur son site. En posant, à propos de la synchronisation et génération des estampilles, les mêmes hypothèses que pour la prévention des deadlocks par estampillage dans les SGBDD, nous arrivons à l'indépendance totale des schedulers locaux. Chacun d'eux peut maintenant décider d'accepter, retarder ou rejeter une opération qui accède à un objet qu'il contrôle de façon autonome, tout comme un scheduler TO centralisé le faisait.

Le DTO, contrairement au D2PL, présente l'intérêt de ne requérir aucune communication entre sites, d'autant plus qu'il prévient naturellement les interblocages.

Toutefois, lorsque le scheduler DTO ainsi défini (c'est-à-dire la version de base) reçoit les opérations dans un ordre fort différent de l'ordre des estampilles, il risque de provoquer un grand nombre d'avortements à cause de sa nature agressive.

12.2. Version conservatrice du DTO

Nous avions suggéré, dans le point consacré à l'estampillage agressif centralisé, une méthode conservatrice extrême qui consistait à exiger de la transaction ses listes d'intention au début de son exécution.

Une autre optique extrêmement conservatrice et que nous présentons maintenant est d'imposer au TM de soumettre ses demandes dans l'ordre des estampilles. L'idée est encore une fois de n'accepter une opération que si on est absolument certain qu'elle pourra toujours s'exécuter jusqu'à la fin.

Le TM générant pour chaque nouvelle transaction une estampille strictement plus jeune que la précédente, il suffit en effet que chaque scheduler gère une file d'attente des opérations reçues mais non encore exécutées. Lorsque le scheduler reçoit une nouvelle opération à exécuter, il l'insère dans sa file en respectant l'ordre des estampilles, et pour des estampilles identiques, l'ordre de réception des demandes. Les transactions les plus âgées se trouveront donc plus près de la sortie de la file d'attente.

Reste encore à régler la manière, pour chaque scheduler, de servir sa file d'attente. Une opération $O_i[X]$ à la sortie de la file d'attente sera dite **prête** (à être exécutée) lorsque :

◊ la file contient au moins une opération en provenance de chaque TM_j du système, et

◊ les exécutions de toutes les opérations déjà envoyées aux DM_j et qui entrent en conflit avec $O_i[X]$ ont été confirmées par les DM_j.

La première règle requiert une file d'attente mémorisant à la fois l'opération stockée et le numéro du TM_j qui l'a soumise. L'usage de ce numéro est en réalité double : il permet de savoir si chaque TM_j a émis une opération et il sert aussi à renvoyer à chacun d'eux la confirmation de l'exécution de l'opération.

La seconde règle pourra être garantie de la même manière que nous l'avions déjà fait pour la version de base de l'estampillage centralisé, à savoir par la gestion de compteurs des lectures et écritures envoyées aux DM_j et non encore confirmées.

La première règle risque cependant de provoquer le blocage de la file d'attente lorsqu' un des TM_j ne transmet plus d'opérations. C'est pourquoi il faut imposer à ces TM qui n'ont plus d'opérations à envoyer d'envoyer une opération conventionnelle qui ne fait rien, NULL par exemple. Mais encore faut-il veiller à ce que l'estampille qui est attribuée à ce NULL soit cohérente avec le protocole imposé[1]. Lorsque cette opération arrive à la sortie

[1] C'est-à-dire garantir que toutes les opérations qui seront envoyées après le NULL auront une estampille supérieure à celle du NULL.

de la file, elle est automatiquement prête, enlevée de la file sans qu'aucun traitement spécial ne lui soit appliqué si ce n'est considérer que le TM_j pour lequel elle a été créée a été servi.

Autre souci amené par la première règle : que se passe-t-il lorqu'un des sites tombe en panne et se trouve incapable d'envoyer une opération (ne serait-ce qu'un NULL) ? Il faudra trouver un moyen et pour signaler aux schedulers qu'un des sites est en panne, et pour signaler qu'un des sites en panne souhaite à nouveau soumettre des opérations.

Comme nous l'avions présagé, ce protocole est extrêmement conservateur puisqu'en fait les transactions sont servies séquentiellement. Sa nature, cependant, permet de réduire le gaspillage de travail dû aux avortements et redémarrages de transactions et par la même occasion, de réduire les échanges de messages. Ce dernier point est heureux dans la mesure où le protocole en lui-même exige la communication de chaque site avec tous les autres, exigence qui devient prohibitive (car exponentielle) lorsque le nombre de sites augmente.

Il existe bien sûr plusieurs méthodes visant à améliorer le degré de concurrence du DTO conservateur, profitant généralement des prédéclarations des transactions lorsque la configuration du système le permet.

Chapitre 13

Test du graphe de sérialisation : DSGT

Rappelons que le principe du test du graphe de sérialisation pouvait s'énoncer comme suit :

◊ construire le graphe de sérialisation, graphe dont les noeuds sont les transactions récemment confirmées ou encore actives et graphe dans lequel un arc entre deux transactions signifie à la fois le conflit et la précédence d'au moins une opération de chaque transaction,

◊ et tester si l'arrivée d'une nouvelle transaction provoque un cycle dans ce graphe; si oui, cette dernière doit être défaite, sinon, elle peut être traitée à condition que toutes les opérations conflictuelles avec celle-ci aient été confirmées par le DM.

Lorsque l'on veut appliquer la méthode du test du graphe de sérialisation dans un environnement distribué où les données n'existent qu'en exemplaire unique, un problème identique à celui de la détection des interblocages distribués se pose : celui d'un ensemble de graphes locaux ne présentant aucun cycle et qui en formeraient un s'ils étaient mis en commun.

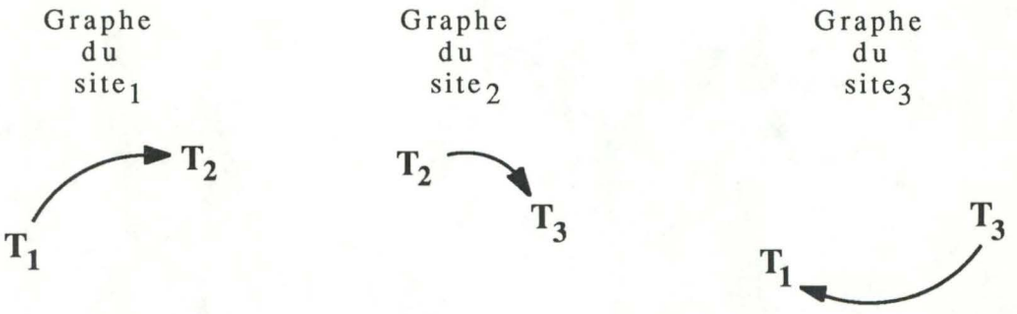
Prenons par exemple les trois transactions T_1, T_2 et T_3 définies ainsi :

T_1	T_2	T_3
TR_BEGIN_1	TR_BEGIN_2	TR_BEGIN_3
$TR\_WRITE_1 (A, V_{11})$	$TR\_READ_2 (A)$	$TR\_READ_3 (B)$
$TR\_WRITE_1 (C, V_{12})$	$TR\_WRITE_2 (B, V_2)$	$TR\_WRITE_3 (C, V_3)$
TR_COMMIT_1	TR_COMMIT_2	TR_COMMIT_3

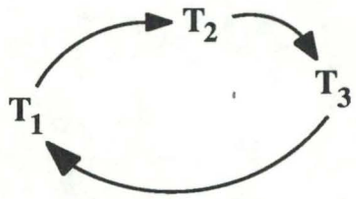
et supposons que les schedulers S_1, S_2 et S_3 exécutent respectivement T_1, T_2 et T_3 comme suit :

$S_1{:}W_1[A]\ S_2{:}R_2[A]\ S_2{:}W_2[B]\ S_3{:}R_3[B]\ S_3{:}W_3[C]$

si maintenant T_1 soumet son $TR\_WRITE_1 (C, V_{12})$ à S_1, celui-ci ne détectant pas de cycle dans son graphe, acceptera de traiter le $W_1[C]$. En effet, les graphes de sérialisation de chaque site sont :



aucun d'eux ne contient de cycles, et pourtant la figure ci-dessous montre bien que globalement, la demande TR_WRITE$_1$ (C, V$_{12}$) en provoque un :



T$_1$ aurait, par conséquent, dû être défaite.

Le problème étant comparable à celui du deadlock global, on serait tenté de lui appliquer le même type de solution. Toutefois, il convient d'être prudent, car bien que les causes soient semblables, les issues sont très différentes.

En effet, tandis que dans une situation de deadlock les transactions sont bloquées jusqu'à ce qu'un mécanisme extérieur intervienne, dans une situation telle qu'illustrée ci-dessus, les transactions ne sont nullement bloquées par le fait du cycle. En conséquence, alors que le mécanisme de détection de deadlock global pouvait espacer ses interventions, celui de détection pour un cycle dans le graphe de sérialisation doit avoir lieu au moins au rythme des transactions.

Le coût de gestion en devient donc prohibitif et le procédé inapplicable pour des applications typiques. Il convient aussi de rappeler la quantité importante de données qu'il fallait traiter dans le SGT centralisé. Cette gêne est encore accentuée dans une configuration répartie et des méthodes pour y remédier deviennent nécessaires. Bernstein et al. [BERN87] consacrent un point particulier à des schedulers visant à minimiser cet espace mémoire.

Chapitre 14

Les méthodes optimistes et le vote

Les hypothèses de travail seront les mêmes que celles déjà adoptées pour les autres méthodes distribuées : chaque site dispose de son certifieur local, responsable de la régulation de l'accès aux données stockées sur ce site. Même si localement, certifieur et DM peuvent fonctionner de façon autonome, il convient évidemment de coordonner l'activité de l'ensemble des certifieurs.

En effet, pour qu'un certifieur puisse valider une transaction T_i,

◊ en certification SGT, il doit avoir obtenu de chaque certifieur ayant reçu des opérations de T_i, son graphe de sérialisation local afin de pouvoir détecter un cycle global[1]. S'il n'existe pas de tel cycle, alors T_i peut être validée; sinon, elle est défaite;

◊ en certification 2PL ou TO, il est nécessaire de procéder à un **vote**. Chaque certifieur local est capable de dire si pour lui, la transaction T_i peut être validée; pour que T_i puisse être globalement validée, il faut que les certifieurs la valident unanimement. Dès qu'un des certifieurs locaux (ne serait-ce qu'un seul) n'a pas pu certifier T_i, elle doit être avortée.

Il est donc exclu qu'un seul certifieur valide une transaction sur base de sa seule décision. Il est également possible qu'il certifie localement une transaction mais que suite au vote, il soit obligé de la défaire.

Sur base de ces deux remarques, si un certifieur invalide une transaction localement, il est certain qu'elle le sera aussi globalement; par contre, s'il la valide, il devra attendre le résultat du vote pour savoir si elle est certifiée globalement.

[1] Nous avons déjà illustré le problème de la détection d'un cycle global lors des points consacrés aux interblocages distribués et lors du test du graphe de sérialisation distribué. Nous estimons superflu de revenir une fois encore sur ce point.

Chapitre 15

Alternatives

15.1. Pour les données non redondantes

Il existe, dans le cadre des SGBDD où les données sont stockées en exemplaire unique, une série de techniques moins conventionnelles que celles sur lesquelles nous nous sommes penchés. Kohler en présente trois dans [KOHL81].

Il existe, surtout dans le domaine des télécommunications (les réseaux locaux par exemple), des techniques dites "à jeton".

Le principe des techniques à jeton est le suivant :

◊ un ensemble de sites sont reliés entre eux de manière à constituer un anneau virtuel dans lequel circule un jeton;

◊ chaque site y gère une transaction (soit de consultation, soit de modification) pour toute sa durée de vie;

◊ le **jeton**, unique, octroie à une transaction d'un site qui le détient, et à elle uniquement, un droit de mise à jour;

◊ une fois cette mise à jour terminée, le jeton est passé au site suivant de l'anneau.

C'est donc le jeton circulant qui assure la sérialisation des mises à jour. L'inconvénient majeur qui en découle est par conséquence une absence totale de concurrence, même lorsque des transactions modifient des données distinctes.

Kohler présente encore deux autres alternatives : l'analyse de conflits et les réservations. Nous passerons sous silence des variantes de cette technique créées pour suppléer au manque de parallélisme des techniques à jeton. Il semblerait en effet que peu de ces méthodes soient vraiment intéressantes, soit à cause de leur faible tolérance aux pannes d'un site, soit à cause de l'exigence de prédéclarations (non applicables à des systèmes interactifs par exemple).

15.2. Pour les données redondantes

Il nous semblait intéressant de présenter dans ce point consacré aux alternatives pour données redondantes divers types de schedulers intégrés. Cette tâche nous est épargnée grâce à un état de l'art en la matière, dressé par Bernstein et Goodman [BERN81]. Ils y décomposent en effet le problème du contrôle de concurrence en deux sous-problèmes : la synchronisation RW (ou WR) et la synchronisation WW, ainsi que nous en avions expliqué les principes dans la première partie de ce travail. Leur article propose en fait un ensemble de 48 méthodes obtenues par composition de différents synchroniseurs : D2PL de base, D2PL avec copies principales, D2PL centralisé, TO de base, TO multiversion, TO conservateur, etc...

Conclusion

Nous avons vu comment les principales méthodes de contrôle de concurrence utilisées dans les BD centralisées pouvaient être récupérées pour un environnement distribué, moyennant quelques modifications destinées à offrir la transparence de localisation et de duplication des informations.

Nous avons dû émettre l'hypothèse que la répartition des sites et des données était résolue. Nous supposons donc qu'il y a un mécanisme de gestion de "catalogues" ou "répertoires" des données sous-jacent au système et qui n'est sans doute pas une tâche facile, puisqu'il doit pouvoir localiser, à tout moment, n'importe quelle donnée, n'importe quelle copie alors que tous ces objets peuvent circuler dynamiquement dans le système.

Tout comme pour la première partie, nous n'avons abordé que les aspects concurrentiels des algorithmes et non pas les problèmes de tolérance aux pannes ou de recouvrement. Dans une configuration répartie cependant, ces problèmes prennent plus d'importance. Une configuration distribuée présente en effet l'avantage d'offrir des possibilités d'accès de site à site et de partage d'informations, mais elle se doit également de présenter l'avantage quelque peu surprenant de l'indépendance des sites, les mettant à l'abri d'éventuelles défaillances d'autres sites.

Cependant, et c'est là un des grands avantages des systèmes distribués, ceux-ci étant par définition composés de plusieurs systèmes indépendants, ils sont globalement plus résistant saux pannes que des systèmes centralisés, où la moindre défaillance risque d'immobiliser tout un système.

Un autre avantage des configurations où une même donnée existe en plusieurs exemplaires est que la disponibilité et donc la sécurité du système sont accrus puisqu'en cas de dommage aux données d'un site, il est encore possible d'accéder aux autres exemplaires des mêmes données stockées en d'autres sites. Toutefois, cet avantage doit être modéré par le fait qu'il faudra aussi assurer l'actualisation de toutes les copies d'un objet, ce que nous avions appelé la "propagation des mises à jour". Il y a donc de grandes chances que ce procédé soit bénéfique pour des lectures mais plus lourd pour des écritures.

Il peut aussi être nécessaire de constituer un environnement distribué afin d'arriver à une capacité suffisante pour une tâche particulière. Un tel système présente de plus l'avantage d'être aisément extensible par l'adjonction d'un nouveau site, par exemple, plutôt que par l'extension d'un site existant.

Finalement, les systèmes distribués sont des configurations lourdes à gérer (par leur taille) mais ils offrent des avantages résultant de la mise en commun de ressources qu'aucun système centralisé ne peut offrir. Nous nous attendons donc à ce que les recherches dans ce domaine s'accentuent et se précisent, étant donné la demande croissante pour de type de configuration.

CONCLUSION

CONCLUSION

Nous avons vu dans ce travail comment les spécificités d'une base de données centralisée, à savoir le grand nombre de ses objets et le caractère imprévisible des clients y accédant, en faisaient une ressource plus difficile à gérer qu'une simple ressource du système d'exploitation, et par conséquent nécessitaient des méthodes nouvelles ou anciennes mais adaptées, et des théories nouvelles, dont la théorie de la sérialisabilité.

Il nous fallait un modèle d'architecture logique du système de gestion de la base de données et dont nous avions reconnu le scheduler comme centre. Ce module concentrait en lui tous les algorithmes de contrôle de concurrence et se trouvait être une interface entre l'utilisateur, représenté par des transactions, et la base de données. Cette interface reconnaissait à l'utilisateur deux grands types d'opérations : les lectures et les écritures, et devait garantir la conservation de la cohérence de la base de données, peu importent les conditions d'utilisation de celles-ci.

La première catégorie d'algorithmes destinés à assurer ce bon fonctionnement des accès était basée sur le verrouillage des objets à accéder. Celui-ci était exclusif ou partagé, selon le type d'opération qui devait être effectuée. Afin d'éviter des problèmes de recouvrement, il fallait imposer un protocole particulier, le verrouillage à deux phases. Celui-ci existait sous plusieurs variantes destinées à combler l'une ou l'autre de ses lacunes, et s'adaptant dès lors mieux à l'une ou l'autre circonstance d'utilisation.

Subsistaient malgré tout deux grandes classes de problèmes inhérents au verrouillage : les interblocages et les famines. Pour les interblocages, il s'agissait d'une situation dans laquelle plusieurs transactions se bloquaient mutuellement en attente d'une donnée qu'une autre détenait dans un mode conflictuel. Nous avons vu deux grandes techniques pour résoudre ces problèmes : la prévention et la détection, toutes deux ayant à leur disposition un graphe des mises en attentes et des outils d'analyse appropriés. Pour les famines, il s'agissait de situations dans lesquelles des transactions étaient privées de ressources sans toutefois se trouver en interblocage. Reposant donc sur une incertitude,

nous ne pouvions leur appliquer des outils mathématiques et devions nous contenter d'une attente suffisamment longue que pour que la famine soit fortement probable.

Autre considération : les verrous pouvaient avoir des "tailles" différentes; ils avaiant tout intérêt à être petits lorsque les modifications étaient localisées et grands lorsqu'elles étaient étalées. Des systèmes de gestion des bases de données fournissaient même des verrous de taille variable, permettant de profiter au maximum du profil des modifications. La gestion des cette variabilité devenait néanmoins coûteuse et ne pouvait s'appliquer qu'à des bases de données structurées physiquement et logiquement d'une manière précise.

Face à tous ces problèmes, s'offraient des solutions sans verrouillage, dont la première était basée sur une attribution de priorités particulières, les estampilles. Celles-ci étaient assignées à chaque transaction qui débutait, et étaient rafraîchies si nécessaire, évitant par là les cas de famine. Cette méthode classait donc les transactions conflictuelles selon l'ordre strict des estampilles et prévenait ainsi également les interblocages. Ces méthodes, existant dans des variantes similaires à celles construites pour le verrouillage, classaient malheureusement les transactions selon un ordre souvent inutilement trop strict.

La deuxième solution sans interblocage était plus orientée vers la théorie des graphes, plus précisément des graphes de sérialisation. Cette solution, moins fréquente que les deux précédentes, était cependant lourde à mettre en oeuvre.

Venaient enfin les méthodes optimistes, appelées ainsi parce qu'elles ne prenaient aucune précaution particulière afin d'éviter les problèmes dus aux conflits sous prétexte que ces situations ne se produisaient que très rarement. Elles se devaient alors de vérifier, aux derniers instants de vie de la transaction, si d'éventuels dégâts n'avaient pas été causés et de les réparer si nécessaire. Ces vérifications pouvaient être menées selon les trois protocoles cités ci-dessus.

Après avoir introduit la multiversion comme nouvelle opportunité pour augmenter le potentiel de concurrence d'accès à la base de données, nous en venions enfin à une évaluation et comparaison des méthodes étudiées, pour en arriver à des méthodes hybrides, scindant le problème du contrôle de concurrence en deux sous-problèmes de synchronisation, des lectures/écritures et des écritures entre elles. Elles visaient à faire résoudre chacun de ces sous-problèmes par une des méthodes déjà vues, mais faisait surgir une grande difficulté qui était d'assurer la symbiose parfaite entre les solutions séparées.

Nous avons à peu près suivi la même démarche pour aborder les bases de données distribuées. Elles se distinguaient en outre des bases de données centralisées de par leur interconnexion en réseau, qui nous amenait à des hypothèses supplémentaires de tolérance aux pannes.

Le modèle de ces configurations était celui d'un ensemble de sites pouvant aussi bien travailler indépendamment que collaborer. Chacun de ces sites avait de plus la possibilité de disposer, dans certaines configurations, d'une partie ou d'un copie des données de la base. Cette partition des sites et des informations, et cette redondance des copies d'une donnée, nous amenaient à considérer les méthodes utilisées pour les bases de données centralisées de façon plus particulière encore.

Ces méthodes, présentées de façon assez succincte, avaient pour objectif de faire apparaître les différences entre les configurations centralisées et les configurations distribuées. Il aurait été intéressant que nous puissions, dans le cadre de ce travail, nous pencher un peu plus sur le problème du contrôle de concurrence pour bases de données réparties, d'autant plus que ce type d'environnement est sans doute promis à un développement important pour les années à venir.

Nous aurions également aimé envisager l'intégration d'un système de gestion de bases de données dans un système d'exploitation, et de comparer cette possibilité avec celle de deux systèmes indépendants. Dans ce cas, les performances auraient certainement été des critères décisifs.

Nous avons enfin envisagé le contrôle de concurrence de manière tout à fait, ou presque, indépendante des problèmes de tolérances aux fautes et de recouvrement. Il serait intéressant d'étudier l'optique contraire, celle d'un contrôle de concurrence et d'un recouvrement étroitement liés.

Il existe dans ces domaines une littérature abondante mais malheureusement fort orientée vers des configurations précises existantes. Peu de ces travaux cherchent à dresser un tableau synthétique de l'existant, de même qu'une explicitation des hypothèses sous-jacentes qui échapent souvent au lecteur moyen. Les différents travaux de Bernstein et al. qui font partie de notre bibliographie sont actuellement les plus louables qui poursuivent ces buts. C'est dans cette direction que nous avons voulu diriger et mener à bien ce travail.

Bibliographie

[AGRA85] AGRAWAL, R., DEWITT, D.J. "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation". ACM Transactions on Database Systems, 10, 4 (December 1985), 529-564.

[BADA86] BADAL, D.Z. "The Distributed Deadlock Detection Algorithm". ACM Transactions on Computer Systems, 4, 4, (November 1986).

[BAYE76] BAYER, R. "On the Integrity of Data Bases and Resource Locking". Data Base Systems, Proc. 5th Int. Symp. IBM Germany, Bad Hombourg. Lecture Notes in Computer Science, 39, 1976, 339-361.

[BAYE80] BAYER, R., HELLER, H., REISER, A. "Parallelism and Recovery in Database Systems". ACM Transactions on Database Systems", 5, 2 (June 1980), 139-156.

[BAYE82] BAYER, R., ELHARDT, K., HEIGERT, J., REISER, A. "Dynamic Timestamp Allocation and its Applications to the BEHR-Method".

[BERN79] BERNSTEIN, P.A., SHIPMAN, D.W., WONG, W.S. "Formal Aspects of Serializability in Database Concurrency Control". IEEE Transactions on Software Engineering, SE-5, 3 (May 1979), 203-216.

[BERN81] BERNSTEIN, P.A., GOODMAN, N. "Concurrency Control in Distributed Database Systems". ACM Computing Surveys, 13, 2 (June 1981), 185-221.

[BERN84] BERNSTEIN, P.A., GOODMAN, N. "An Algorithm for Concurrency Control in Replicated Distributed Databases". ACM Transactions on Database Systems, 9, 4 (December 1984), 596-615.

[BERN87] BERNSTEIN, P.A., HADZILACOS, V., GOODMAN, N. "Concurrency Control and Recovery in Database Systems". Addison-Wesley (1987).

[CARE86] CAREY, M.J., WALEED, A.M. "The Performance of Multiversion Concurrency Control Algorithms". ACM Transactions on Computer Systems, 4, 4, (November 1986).

[CHAN87] CHANTEUX, C., THIRY, J. "Tolérance aux pannes dans les systèmes informatiques de gestion". Mémoire de fin d'études (1987).

[CRAF83] CRAFT, D.H. "Resource Management in a Decentralized System" Proceedings of the Second ACM Symposium on Principles of Distributed Computing, (August 1983), 11-19.

[CURT77] CURTICE, R.M. "Integrity in Database Systems". Datamation (May 1977), 64-68.

[DATE83] DATE, C.J. "An Introduction to Database Systems". Addison-Wesley (1983).

[EAGE83] EAGER, D.L., KENNETH, C.S. "Achieving Robustness in Distributed Database Systems". ACM Transactions on Database Systems, 8, 3, (September 1983), 354-381.

[ESWA76] ESWARAN, K.P., GRAY, J.N., LORIE, R.A., TRAIGER, I.L. "The Notions of Consistency and Predicate Locks in a Database System". Communications of the ACM, 19, 11 (November 1976), 624-633.

[FRAN85] FRANASZEK, P., ROBINSON, J.T. "Limitations of Concurrency in Transaction Processing". ACM Transactions on Database Systems, 10, 1 (March 1985), 1-28.

[GOOD85] GOODMAN, N., RAJAN, S., TAY, Y.C. "Locking Performance in Centralized Databases". ACM Transactions on Database Systems, 10, 4, (December 1985), 415-462.

[GRAY79] GRAY, J.N. "Notes on Data Base Operating Systems". In Operating Systems : An advanced course, R. BAYER, R.M. GRAHAM, and G. SEEGMULLER, Eds., Springer-Verslag, New York, 1979, 393-481.

[HAIN88] HAINAUT, J.L. "Technologie des fichiers : Systèmes de gestion de fichiers et bases de données". Notes de cours (1988).

[HERL87] HERLIHY, M. "Concurrency versus Availability. Atomicity Mechanisms for Replicated Data". ACM Transactions on Computer Systems, 5, 3, (August 1987), 249-274.

[KOHL81] KOHLER, W.H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems". ACM Computing Surveys, 13, 2 (June 1981), 149-183.

[KUNG81] KUNG, H.T., ROBINSON, J.T. "On Optimistic Methods for Concurrency Control". ACM Transactions on Database Systems, 6, 2 (June 1981), 213-226.

[LAMP85] LAMPORT, L. "Solved Problems, Unsolved Problems and Non-Problems in Concurrency". ACM Operating Systems Review, 19, 4, (October 1985).

[LAUS85] LAUSEN, G., SOISALON-SOININEN, E., WIDMAYER, P. "Pre-analysis Locking : A Safe and Deadlock Free Locking Policy". Proceedings of VLDB 85, Stockholm, 270-281.

[MENA80] MENASCE, D.A., POPEK, G.J., MUNTZ, R.R. "A Locking Protocol for Ressource Coordination in Distributed Databases". ACM Transactions on Database Systems, 5, 2, (June1980), 103-138.

[MOHA82] MOHAN, C., FUSSEL, D., SILBERSCHATZ, A. "A Biased Non-Two-Phase Locking Protocol". Department of Computer Sciences, University of Texas at Austin, 337-361.

[MOHA85] MOHAN, C., FUSSEL, D., KEDEM, Z.M., SILBERSCHATZ, A. "Lock Conversion in Non-Two-Phase Locking Protocols". IEEE Transactions on Software Engineering, SE-11, 1 (January 1985), 15-22.

[OBER82] OBERMARCK, R. "Distributed Deadlock Detection Algorithm". ACM Transactions on Database Systems, 7, 2, (June 1982), 187-208.

[PEIJ87] PEI-JYUN, L., BHARAT, B. "Multidimensional Timestamp Protocols for Concurrency Control". IEEE Transactions con Software Engineering, SE-13, 12, (December 1987), 1238-1253.

[PETE83] PETERSON, J.L., SILBERSCHATZ, A. "Operating System Concepts". Addison Wesley (August 1983), 257-385.

[RIES77] RIES, D.R., STONEBRAKER, M. "Effects of Locking Granularity in a Database Management System". ACM Transactions on Database Systems, 2, 3 (September 1977), 233-246.

[RYPK79] RYPKA, D.J., LUCIDO, A.P. "Deadlock Detection and Avoidance for Shared Logical Resources". IEEE Transactions on Software Engineering, SE-5, 5, (September 1979), 465-471.

[SCHL78] SCHLAGETER, G. "Process Synchronisation in Database Systems". ACM Transactions on Database Systems, 3, 3, (September 1978),248-271.

[SPEC85] SPECTOR, A.Z., DANIELS, D., DUCHAMP, D., EPPINGER, J.L., PAUSCH, R. "Distributed Transactions for Reliable Systems". ACM Transactions on Database Systems, (December 1985), 127-146.

[STON84] STONEBRAKER, M. "Virtual Memory Transaction Management". ACM Operating Systems Review, 18, 2, (April 1984).

[SINH85] SINHA, M.K., NATARAJAN, N. "A Priority Based Distributed Deadlock Detection Algorithm". IEEE Transactions on Software Engineering, SE-11, 1, (January 1985), 67-80.

[TRAI82] TRAIGER, I.L., GRAY, J., GALTIERI, C.A., LINDSAY, B.G. "Transactions and Consistency in Distributed Database Systems". ACM Transactions on Database Systems, 7, 3, (September 1982), 323-342.

[WEIH83] WEIHL, W.E. "Data-dependent Concurrency Control and Recovery". Proceedings of the Second ACM Symposium on Principles of Distributed Computing, (August 83), 19-31.

[WEIN85] WEINSTEIN, M.J., PAGE, T.W., LIVEZEY, B.K., POPEK, G.J. "Transactions and Synchronization in a Distributed Operating System". ACM Operating Systems Review, 19, 5, (December 1985), 115-126.

[WIED83] WIEDERHOLD, G. "Database Design". Computer Science Series (1983), 613-645.